A Foundational Proof Framework for Cryptography

A DISSERTATION PRESENTED
BY
ADAM PETCHER
TO
THE SCHOOL OF ENGINEERING AND APPLIED SCIENCES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN THE SUBJECT OF
COMPUTER SCIENCE

HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS
MAY 2015
Distribution A: Public Release

©2014 – Adam Petcher
all rights reserved.

This work is sponsored by the United States Air Force and the Intelligence Advanced Research Projects Activity under Air Force Contract #FA8721-05-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the author and are not necessarily endorsed by the United States Government.

Thesis advisor: Professor Greg Morrisett Adam Petcher

A Foundational Proof Framework for Cryptography

Abstract

I present a state-of-the-art mechanized framework for developing and checking proofs of security for cryptographic schemes in the computational model. This system, called the Foundational Cryptography Framework (FCF) is based on the Coq proof assistant, and it provides a sophisticated mechanism for reasoning about cryptography on top of a simple semantics and a small trusted computing base. All of the theory and logic of FCF is proved correct within Coq, thus ensuring that all security results are trustworthy. FCF improves the state of the art by providing a fully foundational system that enjoys the same ease of use of current non-foundational systems.

Facts proved using FCF include the security of El Gamal encryption, HMAC, and an efficient searchable symmetric encryption (SSE) scheme. The proof related to the SSE scheme is among the most complex mechanized cryptographic proofs, and this proof demonstrates that FCF can be used to prove the security of complex schemes in a foundational manner.

FCF provides a language for probabilistic programs, a theory that is used to reason about programs, and a library of tactics and definitions that are useful in proofs about cryptography. Proofs provide concrete bounds as well as asymptotic security claims. The framework also includes an operational semantics that can be used to reason about the correctness and security of implementations of cryptographic systems.

Contents

1 INTRODUCTION 1

2 BACKGROUND 4
- 2.1 The Coq Proof Assistant . . . 5
- 2.2 Proofs in Cryptography . . . 6
- 2.3 Mechanized Frameworks for Cryptographic Proofs . . . 9

3 FRAMEWORK DESIGN 12
- 3.1 Design Goals . . . 13
- 3.2 Framework Introduction . . . 14
- 3.3 Cryptographic Arguments in FCF . . . 21
- 3.4 Comparison . . . 24
- 3.5 Conclusion . . . 28

4 TECHNICAL DESCRIPTION 30
- 4.1 Probabilistic Programs . . . 32
- 4.2 Theory of Distributions . . . 35
- 4.3 Program Logic . . . 38
- 4.4 Asymptotic Theory . . . 40
- 4.5 Operational Semantics and Reasoning about Code . . . 43
- 4.6 Related Work . . . 45
- 4.7 Conclusion . . . 47

5 EXAMPLE PROOFS 49
- 5.1 El Gamal Encryption . . . 50
- 5.2 Symmetric Encryption from a Pseudorandom Function . . . 58
- 5.3 A Negative Example: Dual_EC_DRBG . . . 68
- 5.4 Conclusion . . . 71

6 SEARCHABLE SYMMETRIC ENCRYPTION 72
6.1 Searchable Symmetric Encryption Proof Overview 74
6.2 Single Keyword Searchable Symmetric Encryption from Tuple Sets 75
6.3 Tuple Set Instantiation 82
6.4 Proof Engineering 95
6.5 Related Work 98
6.6 Conclusion 99

7 PROVABLY SECURE IMPLEMENTATIONS 100
7.1 Extracting Code from FCF Models 101
7.2 Verifying C Code 103
7.3 Related Work 110
7.4 Conclusion 112

8 SUMMARY AND CONCLUSION 113
8.1 Choosing a Cryptographic Proof Framework 114
8.2 Future Work 115

APPENDIX A ADEQUACY OF OPERATIONAL SEMANTICS 118
A.1 The Value of Adequacy 119
A.2 Adequacy Theorem 120
A.3 Conclusion 126

REFERENCES 130

Listing of figures

2.1 Semantic Security Game 7

4.1 Semantics of Probabilistic Computations 32
4.2 Semantics of Computations with Oracle Access 35
4.3 Small-step Operational Semantics 44

5.1 PRF Encryption Sequence of Games 61

6.1 SSE Security Proof Structure 75
6.2 Single-Trial T-Set Security Games 87
6.3 Illustration of "Many to Core" Argument 89
6.4 Single-Trial T-Set Correctness Games 93

7.1 HMAC Security Proof Structure 109

1

Introduction

Cryptographic algorithms and protocols are becoming more numerous, specialized, and complicated. The security of these schemes is traditionally ensured by the development of a mathematical proof of security, or by widespread efforts to find weaknesses. The latter approach is probably impractical for specialized systems, and the former approach suffers from the issue that many of these proofs are not carefully verified. To address this problem, some cryptographers[15,34] have proposed an increased level of rigor and formality for cryptographic proofs. The ultimate goal of this formality is the development of a system that allows cryptographers to describe cryptographic schemes and security proofs using a formal language that allows the proofs to be checked automatically by a highly trustworthy mechanized proof checker.

To enable such mechanically-verified proofs, I have developed The Foundational Cryptography Framework (FCF). This framework embeds into the Coq proof assistant[37] a simple probabilistic programming language to allow the specification of cryptographic schemes, security definitions, and assumptions. The framework also includes useful theory, tactics, and definitions that assist with the construction of proofs of security. Once complete, the proof can be checked by the Coq proof checker. FCF improves on existing tools for checking cryptographic proofs by significantly increasing the trustworthiness of the result and providing other desirable features such as integration with Coq and reasoning about implementations.

This paper is organized as follows: I begin by providing some background (Chapter 2) on cryptographic proofs and the technology used to mechanize them. Then I explain the design of FCF (Chapter 3) and introduce the proof development process using a number of simple examples. Chapter 4 provides a complete technical and theoretical description of FCF.

I developed several example proofs in FCF in order to exercise the framework and provide information on how to develop such proofs, and these are described in Chapter 5. An important consideration for a mechanized cryptography framework is the degree to which the framework and proof techniques scale to proofs about complex systems. To demonstrate the scalability of FCF, I completed a mechanized proof of security for a complex searchable symmetric encryption scheme (Chapter 6). FCF was designed to support reasoning about implementations of cryptographic systems as well as models of cryptographic schemes. In Chapter 7 I describe the process of verifying implementations of cryptographic schemes, including an effort that produced a mechanized proof of security for an efficient implementation of HMAC.

Finally, I summarize the current state of the art of mechanized cryptographic proofs in Chapter 8 and suggest some courses for future work.

2

Background

FCF builds on a large amount of existing work in the fields of formal reasoning tools and cryptography. In this chapter, I provide some background information on the Coq proof assistant, proofs in cryptography, and existing tools and frameworks for formal reasoning about cryptography.

2.1 THE COQ PROOF ASSISTANT

Coq is a proof assistant that can be used to develop and check mathematical proofs. This system includes a language called Gallina for specifying definitions, algorithms, statements and proofs. The process of writing a proof in Gallina is somewhat unnatural, so Coq also includes a language called Ltac which allows the developer to construct proofs in a more natural way by a applying a sequence of tactics.

A simple example is provided to familiarize the reader with Coq. Listing 1 uses Coq's *Inductive* mechanism to define the set of natural numbers. Listing 2 contains a recursive definition of "less than or equal to" (≤) for natural numbers. Listings 3 and 4 contain proofs that ≤ is reflexive and transitive, respectively. These proof proceed by induction on the set of natural numbers, which is possible because Coq automatically produced an induction principle from the inductive definition of natural numbers.

```
(* Data types are defined inductively *)
(* A natural number is either zero or the
   successor of a natural number *)
Inductive Natural :=
  | zero : Natural
  | successor : Natural -> Natural.
```

Listing 1: Inductive Data Type for Natural Numbers

```
(* NatLE is a function that defines what it means
   for a Natural to be less than or equal to
   another Natural.  Prop (Proposition) is the
   type of Coq statements.  *)
Fixpoint NatLE(n1 n2 : Natural) : Prop :=
  match n1 with
    | zero =>  True
    | successor n1' =>
      match n2 with
        | zero => False
        | successor n2' =>
          NatLE n1' n2'
      end
  end.
```

Listing 2: ≤ for Natural Numbers

```
Theorem NatLE_refl : forall n1,
  NatLE n1 n1.

  (* induction on n1 *)
  induction n1.

  (* base case: NatLE zero zero *)
  (* simplify this term to get True *)
  simpl.
  (* True is trivially true *)
  trivial.

  (* step case:
      NatLE (successor n1) (successor n1) *)
  (* simplify this term to get NatLE n1 n2 *)
  simpl.
  (* apply induction hypothesis *)
  apply IHn1.

Qed.
```

Listing 3: Reflexivity of $\leq$

```
Theorem NatLE_trans : forall n1 n2 n3,
  NatLE n1 n2 ->
  NatLE n2 n3 ->
  NatLE n1 n3.

  (* induction on n1, then destruct other terms *)
  (* intuition splits goals and discharges
      trivial ones *)
  induction n1; intuition; simpl in *.
  destruct n2; intuition; simpl in *.
  destruct n3; intuition; simpl in *.

  (* automatically apply induction hypothesis *)
  eauto.
Qed.
```

Listing 4: Transitivity of $\leq$

2.2 PROOFS IN CRYPTOGRAPHY

Proofs in cryptography are typically given in the form of a reduction that proves the security of some scheme or construction assuming some other problem (or set of problems) is hard for a computationally-bounded adversary to solve. If I want to prove that scheme S is secure assuming that problem T is hard, I start by assuming that there is some adversary A that can *effectively* defeat the security of scheme S. Then I use A to construct a procedure B that can *effectively* solve problem T. In doing so, I have produced a contradiction, and the initial assumption of the existence of A must be false.

The desired notion of security of a cryptographic scheme is expressed using “games”, in which an adversary is required to interact with the scheme in a particular way. A game produces a bit which is used to determine whether the adversary wins the game. For example, semantic security (Figure 2.1) is a desirable property of encryption schemes in which the adversary chooses a plaintext and is given either the corresponding ciphertext or the encryption of some constant value. The adversary produces a bit to indicate whether he was given a ciphertext corresponding with his chosen plaintext, and he wins the game if this bit is correct. Security definitions may be given in the form of a single game, as in the semantic security game in Figure 2.1, or in the form of a pair of games that that the adversary should be unable to distinguish. In the corresponding semantic security definition using two games, the adversary is given the encryption of his selected plaintext in one game and the encryption of the constant value in another. In both games, the adversary produces a bit, and he wins if this bit is noticeably different in the two games. Note that, in both cases, the definitions are concerned with the *distributions* on the bits produced by the games.

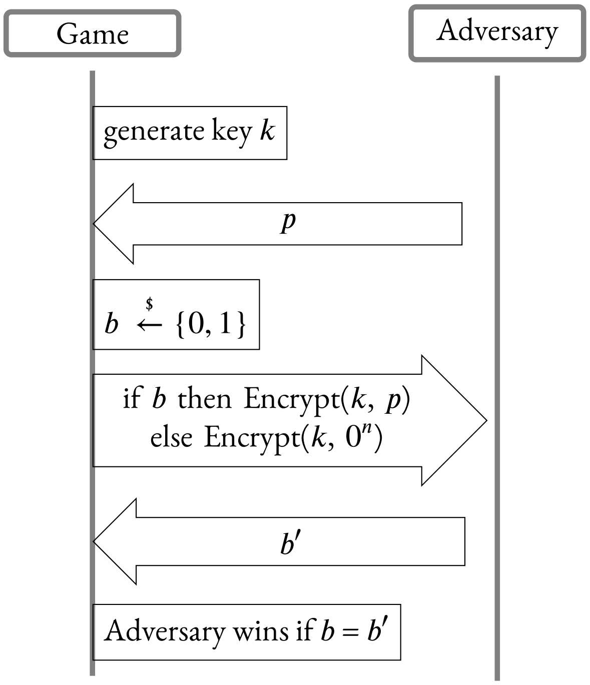

Game
Adversary
generate key k
p
$b \xleftarrow{\$} \{0,1\}$
if b then Encrypt(k, p)
else Encrypt(k, 0^n)
b'
Adversary wins if $b = b'$

Figure 2.1: Semantic Security Game

The “effectiveness” of an adversary must be carefully measured in order for a proof to be meaningful. Effectiveness has two components: the resources available to the adversary and the probability that the adversary wins the game. In the computational model, the resource available to the adversary is a limited amount of running time, but other models limit the storage used, the number of oracle queries allowed, or any other resources.

A traditional proof of security describes a *family* of schemes and adversaries indexed by a natural number η. For example, an encryption scheme may support keys of length η for any value of η, and the security of the scheme is expected to increase as η increases. In this setting the resources and success probability of the adversary can be determined as functions of η. Typically, the scheme is secure if any adversary with an amount of resources that is *polynomial* in η (e.g. probabilistic polynomial time) has *negligible* probability of winning the game.

It is often helpful to prove the *exact security* of some cryptographic scheme. That is, the probability of an adversary winning the security game is given as an expression. This expression may include η (if applicable) or the parameters describing the resources available to the adversary. In this setting, assumptions related to the hardness of certain problems show up as terms in this expression. For example, a bound on the probability that at an adversary defeats an encryption scheme may be a sum, where the first term is the probability that some other (constructed) adversary is able to distinguish a pseudorandom function from a random function, and the second term is the probability of a (highly unlikely) collision. In the case of this example, this expression must be inspected to conclude that it is “sufficiently small” assuming that the first term is small. It is possible to derive asymptotic claims from these concrete bounds, but they are also very valuable in practice, since they provide precise guidance for selecting system parameters in order to obtain the desired level of security.

A popular method for developing and expressing cryptographic proofs is the “sequence of games” style[15]. Instead of directly proving that some probability value is small or that two of these values are "close", I can develop a sequence of games and prove that each game in the sequence is appropriately related to the game that precedes it. The goal is to use this sequence to transform some initial game into a game that obviously has some desired property (e.g. it corresponds to a small probability value or it exactly equals some other game in a security definition). The relation on a pair of games may indicate that the games correspond to identical distributions, that some probability value is less than another, or the probability values are separated by at most some "small" value. These proofs can be more manageable since each pair of games corresponds to a very small transformation, and each of these transformations can be inspected individually. This style of proof can provide exact security results, since the final expression can be determined by summing the non-zero distances between pairs of games.

The "sequence of games" style is ideal for formal reasoning about cryptographic proofs, because it can be used to divide a complex proof into several smaller reasoning steps. Each of these steps is relatively simple because only the transformation in question must be considered, and the detail associated with the rest of the cryptographic scheme and proof can be ignored. As a result of this simplification, the search space is greatly reduced, and proof search (performed either by a human or an automated tool) is expedited. A significant benefit of mechanized proofs in this style is that the sequence of games does not need to be trusted or inspected—it is merely a tool used to develop the final result of the proof.

2.3 MECHANIZED FRAMEWORKS FOR CRYPTOGRAPHIC PROOFS

Several mechanized systems have been developed to check cryptographic proofs in the "sequence of games" style.

CryptoVerif[20] was one of the first systems for reasoning about cryptographic proofs in the computational model. This system is completely automated, and it can even produce the sequence of games from a model of the construction and the desired security property. CryptoVerif is very limited in the sorts of constructions and security properties that it supports. Notably, the tool only supports security properties related to secrecy and authenticity. As a result, CryptoVerif cannot reason about many interesting areas of cryptography including foundations (e.g. pseudorandom functions, oblivious transfer), certain applications (e.g. multiparty computation, zero-knowledge proofs), and even variations on schemes that provide secrecy or authenticity (e.g. searchable/homomorphic/functional encryption). The language of CryptoVerif is also limited because it does not contain loops. This limitation is necessary to support automation, but it prevents CryptoVerif from reasoning about constructions that require certain forms of looping behavior.

The first fully-general system for reasoning about cryptography was CertiCrypt[11], which was later followed by EasyCrypt[9]. CertiCrypt is a framework that is built on Coq, and allows the development of mechanized proofs of security in the computational model for arbitrary cryptographic constructions. Unfortunately, proof development in CertiCrypt is time-consuming, and the developer must spend a disproportionate amount of time on simple, uninteresting goals. To address these limitations, the group behind CertiCrypt developed EasyCrypt, which has a similar semantics and logic, and uses the Why3 framework and SMT solvers to improve proof automation. EasyCrypt takes a huge step forward in terms of usability and automation, but it sacrifices some trustworthiness due to that fact that the trusted computing base is larger and the basis of the mechanization is a set of axiomatic rules.

EasyCrypt represents the state-of-the-art in general-purpose frameworks for formally reasoning about cryptographic schemes. This system has several limitations, though, and chief among them is its lack of a mechanism to extend the tool in a trustworthy manner. Extensibility is crucial to the viability of a cryptographic framework because the framework must be able to handle new sorts of constructions and theory, and it must support new methods of reasoning about the behavior of constructions. FCF was designed to provide the “ease of use” of EasyCrypt combined with a trustworthy mechanism to extend the framework and a generally increased level of trustworthiness.

3

Framework Design

In this chapter, I describe the design goals of FCF and introduce the framework using a series of examples. Since FCF was designed to combine the usability of EasyCrypt with an increased level of trustworthiness, I will also compare FCF to EasyCrypt with respect to these design goals.

3.1 DESIGN GOALS

Based on my experience working with EasyCrypt, I formulated a set of idealized design goals that a practical mechanized cryptography framework should satisfy.

Familiarity. Security definitions and descriptions of cryptographic schemes should look similar to how they would appear in cryptography literature, and a cryptographer with no knowledge of programming language theory or proof assistants should be able to understand them. Furthermore, a cryptographer should be able to inspect and understand the foundations of the framework itself.

Proof Automation. The system should use automation to reduce the effort required to develop a proof. Ideally, this automation is extensible, so that the developer can produce tactics for solving new kinds of goals.

Trustworthiness. Proofs should be checked by a trustworthy procedure, and the core definitions (*e.g.*, programming language semantics) that must be inspected in order to trust a proof should be relatively simple and easy to understand.

Expressivity. It should be possible to express any known cryptographic security definition, construction, or model in the language of the framework. Further, the framework should be able to check a mechanized form of any cryptographic proof.

Extensibility. It should be possible to directly incorporate any existing theory that has been developed for the proof assistant. For example, it should be possible to directly incorporate an existing theory of lattices in order to support cryptography that is based on lattices and their related assumptions. The framework should also support trustworthy addition of new theory for reasoning about the behavior of cryptographic constructions.

Concrete Security. The security proof should provide concrete bounds on the probability that an adversary is able to defeat the scheme. Concrete bounds provide more information than asymptotic statements, and they inform the selection of values for system parameters in order to achieve the desired level of security in practice.

Abstraction. The system should support abstraction over types, procedures, proofs, and modules containing any of these items. Abstraction over procedures and primitive types is necessary for writing security definitions, and for reasoning about adversaries in a natural way. The inclusion of abstraction over proofs and structures adds a powerful mechanism for developing sophisticated abstract arguments that can be reused in future proofs.

Secure Implementations. The system should be able to reason about the security of implementations of cryptographic systems. The implementation could be produced by extracting code from a model, or by proving that some code is equivalent to the model.

3.2 Framework Introduction

This section provides a brief introduction to the Foundational Cryptography Framework. FCF is explained by example, and all of the examples in this section are elements of larger proofs described in later chapters.

3.2.1 Probabilistic Programs

FCF provides a common probabilistic programming language for describing all cryptographic constructions, security definitions, and problems that are assumed to be hard. Probabilistic programs are described using Gallina, the purely functional programming language of Coq, extended with a computational monad that adds sampling uniformly random bit vectors. The type of probabilistic computations that return values of type `A` is `Comp A`. The code uses `{0,1}^n` to describe sampling a bit vector of length `n`. Arrows (e.g. `<-$`) denote sequencing (i.e. bind) in the monad. Other notation used in the listings will be described when its meaning is not apparent.

Listing 5 contains an example program implementing a one-time pad on bit vectors of length

```
Definition OTP c (x : Bvector c) : Comp (Bvector c)
  := p <-$ {0, 1}^c; ret (BVxor c p x)
```

Listing 5: Example Program: One-Time Pad

`c` (for any natural number `c`). The program produces a random bit vector and stores it in `p`, then returns the *xor* (using the standard Coq function `BVxor`) of `p` and the argument `x`.

3.2.2 Semantics and Probability Theory

The language of FCF has a denotational semantics that relates programs to discrete, finite probability distributions. A distribution on type A is modeled as a function in $A \rightarrow \mathbb{Q}$ which should be interpreted as a probability mass function. This semantics can be used to show that the probabilities of two events are equal, related by an inequality, or distant by at most some value. All of these claims are necessary in order to complete proofs in the "sequence of games" style, in which several games are provided, and relations on adjacent pairs of games are proven. The semantics can also be used to determine an exact value for the probability of an event, which is necessary to provide concrete bounds in security proofs.

FCF provides a theory of distributions that can be used to complete proofs without appealing directly to the semantics. FCF also provides a library of tactics that apply individual theorems, sequences of theorems, or perform non-trivial computations in order to discharge goals. The theory is all proven in Coq from the semantics, and the tactics only apply theorems, so these objects are not in the trusted computing base of FCF.

Using the theory and tactics, I can complete proofs as shown in Listing 6. In this proof, I show that a one-time pad applied to an arbitrary value has the same distribution as a random bit vector. In the statement of the theorem, `D` represents the denotational semantics, which is used to obtain the distribution corresponding to the program that follows it. Because these distributions are represented as functions, I compare them with respect to an arbitrary value `y` in the distribution. I use the notation `Pr[c]` to represent the probability that Boolean computation c produces `true`. The `==` symbol represents equality for rational numbers.

The proof proceeds by using tactics to transform the goal or hypotheses until I get a goal that is trivial and can be automatically discharged. I use `intuition` to introduce all variables, then I `unfold` the definition of `OTP` to replace `D(OTP x)` with the body defined in Listing 5. `r_ident_r` is an FCF tactic that uses Coq's *rewrite* tactic along with a monadic right identity theorem to replace `D({0,1}^c)` with `D(a <-$ {0,1}^c; ret a)`. This transformation puts the goal into a form where we can apply the distribution isomorphism theorem (Theorem 4 in Chapter 4) to complete the proof. At a high level, this theorem allows us to prove that two distributions are equivalent by showing that there is a bijection on the supports of the distributions that preserves the probability mass of the corresponding values. The theorem takes a bijection and its inverse, and we supply the involution `(BVxor c x)` for both. When this theorem is applied, several simpler goals are produced. These goals are either trivial equalities or simple facts about the `BVxor` function (e.g. commutativity, identity) which can be discharged by the specialized `xorTac` tactic.

```
Theorem OTP_eq_Rnd:
  forall (x y : Bvector c),
    D (OTP x) y == D ({0, 1}^c) y.

    intuition. unfold OTP.
    r_ident_r.
    eapply (dist_iso (BVxor c x) (BVxor c x));
      intuition; xorTac.
  Qed.
```

Listing 6: Example Proof: Equivalence of One-Time Pad

Once I have proven the theorem in Listing 6 I can use this theorem to rewrite anything that unifies with either expression. I can also use other theorems and tactics to focus on some location in the program and perform this rewrite at that location. The ability to perform such rewrites provides the basis for completing proofs composed of sequences of games.

The language of FCF also includes a `(Repeat c P)` statement that repeats computation `c` until a decidable predicate `P` holds on the result. This is equivalent to conditioning the distribution corresponding to `c` on the event `P`.

A simple program that uses `Repeat` to sample uniformly-distributed natural numbers in $[0, n)$ is shown in Listing 7. `RndNat_h` is a helper function that samples a natural number with the appropriate number of bits. In this function, `lognat` computes the base-2 logarithm (rounded down) of the argument and `bvToNat` converts a bit vector to the corresponding natural number. The `RndNat` procedure repeats `RndNat_h` until the result is less than `n`, as determined by the function `ltNat`. It is possible to show that this procedure corresponds with a uniform distribution on numbers in the specified range, and this theorem is present in the FCF library.

```
Definition RndNat_h(n : nat) :=
  v <-$ {0,1} ^ (lognat n); ret (bvToNat v).

Definition RndNat(n : nat) :=
  (Repeat (RndNat_h n) (fun x => (ltNat x n))).
```

Listing 7: Example Program: Random Natural Numbers

3.2.3 PROGRAM LOGIC

Many proofs can be completed using the theory of distributions alone, but it can be difficult to complete a proof involving state or looping behavior in this manner. To assist with such proofs, FCF includes a program logic in the style of EasyCrypt. The program logic allows relational judgments on pairs of probabilistic programs. The syntax of a judgment is `(comp_spec P c1 c2)`, indicating that relational predicate P holds (probabilistically) on the values produced by programs `c1` and `c2`. A more detailed description of the program logic is provided in Chapter 4.

Listings 8 and 9 illustrate the program logic using the `compMap` construction, which maps a computation over a list. This function uses Coq's `Fixpoint` to destruct the list and apply the computation to the first element, then recursively call `compMap` on the remainder of the list.

The `compMap_rel` theorem describes a relational program logic judgment for this construction. This judgment requires that some predicate `P1` holds on all corresponding pairs of values in lists `lsa` and `lsb` (defined using Coq's `Forall2`). Additionally, for any pair of values `a` and `b` on which `P1` holds, the relation `P2` must hold on `(c1 a)` and `(c2 b)`. Then the theorem states that `P2` holds on all corresponding pairs of values in the lists resulting from the map operation.

The relational program logic is a powerful tool for completing proofs of security involving sequences of games. In such a proof, it is necessary to prove that some relation holds on each adjacent pair of games in the sequence. The program logic provides a general mechanism for proving that arbitrary relations hold on subprograms appearing within those games. These judgments can be combined to prove judgments on the entire games, including judgments that correspond to equality, inequality, and closeness of probability distributions.

The `compMap_fission` theorem is another judgment on `compMap` describing equivalence of loop fission. Various forms of this theorem, along with similar theorems for probabilistic fold operations, are used extensively in the proofs in Chapter 6. This theorem can be proved by induction on the list using existing program logic facts and tactics.

```
Fixpoint compMap(c : A -> Comp B)(ls : list A) :
    Comp (list B) :=
  match ls with
    | nil => ret nil
    | a :: lsa' =>
      b <-$ c a;
      lsb' <-$ compMap c lsa';
      ret (b :: lsb')
  end.
Theorem compMap_fission :
  forall (c1 : A -> Comp B)(c2 : B -> Comp C)
    (ls : list A),
    comp_spec eq
      (compMap (fun a => b <-$ c1 a; c2 b) ls)
      (ls' <-$ compMap c1 ls; compMap c2 ls').
```

Listing 8: Probabilistic Map and Fission Equivalence

```
Theorem compMap_rel :
  forall (P1 : A -> B -> Prop)(P2 : C -> D -> Prop)
         (lsa : list A)(lsb : list B)
         (c1 : A -> Comp C)(c2 : B -> Comp D),
    Forall2 P1 lsa lsb ->
    (forall a b, In a lsa -> In b lsb ->
      P1 a b -> comp_spec P2 (c1 a) (c2 b)) ->
    comp_spec (Forall2 P2)
                (compMap c1 lsa)
                (compMap c2 lsb).
```

Listing 9: Relational Judgment on Probabilistic Map

3.2.4 Computations with Oracle Access

It is common for a security definition to include some notion of state. For example, the adversary may comprise multiple procedures that are allowed to share state. In this case, the state can be passed explicitly or using a state monad. This solution is not sufficient in all circumstances, though. Consider a security definition in which an adversary is allowed to query an oracle that must maintain state across calls to the oracle. If the state monad was used, then the adversary would be able to inspect or modify the state of the oracle. To address this issue, FCF includes a type for a procedure that has access to a stateful oracle. This type is given a semantics that allows the procedure to query the oracle without being able to view or modify the state of the oracle. Using this type, I can create adversary/oracle interactions such as the one shown in listing 10. This game, which is part of an oracle-based semantic security definition, chooses an encryption key at random and then creates an oracle that uses that key to encrypt any plaintexts it receives. The adversary procedure `A` has the type of a procedure with oracle access. When `A` is applied to an oracle and an initial state, a coercion invokes the semantics associated with the type of `A`, producing an interaction that prevents `A` from accessing the state of the oracle. The result is a computation that produces a pair: the first value is the output of the adversary, and the second value is the final state of the oracle.

```
Definition IND_CPA_SecretKey_O_G0 :=
  key <-$ KeyGen ;
   [b, _] <-$2 A (EncryptOracle key) tt;
   ret b.
```

Listing 10: Example Adversary/Oracle Interaction

3.2.5 TACTICS

The most commonly used theorems in the theory of distributions and the program logic have tactics associated with them that make them easier to apply. In many cases, a theorem related to distributions has a corresponding theorem in the program logic, and a single tactic can be used to apply the appropriate form of the theorem based on the current goal. For example, the `comp_skip` tactic will apply the distribution isomorphism theorem introduced in Listing 6, using the identity function as the bijection. This tactic has the effect of simply removing identical pairs of statements at the beginning of the games, and this tactic can be successfully invoked when the goal is either an (in)equality of distributions or a program logic judgment.

All of the primitive tactics like `comp_skip` apply to the beginning of the games. A tactical called `comp_at` can be invoked to apply any primitive tactic at an arbitrary position within a game. There are also slightly more sophisticated tactics, such as `inline_first` which extracts the first statement in a deeply nested computation, `comp_simp` which simplifies programs, and `dist_compute` which performs case splits and other manipulations in order to compute a numeric probability value corresponding to a simple program.

3.2.6 Programming Library

FCF includes a library that includes several standard programming constructs and their associated theory. This library includes the `compMap` operation seen in Listing 8 as well as other list operations such as probabilistic fold and summation. This package uses the program logic extensively, and many of the theorems take a specification on a pair of computations as an argument, and produce a specification on the result of folding/mapping those computations over a list. The package also contains theorems about typical list and loop manipulations such as appending, flattening, fusion and order permutation.

The library also includes additional constructed sampling routines such as sampling from lists, groups, and arbitrary Bernoulli distributions with rational success probability. These sampling routines are all computations based on the `Rnd` statement provided by the language, and each routine is accompanied by a theory establishing that the resulting distribution is correct.

3.2.7 Operational Semantics

FCF also provides a conventional operational semantics for its language in order to allow extraction of OCaml programs from FCF constructions as well as relating FCF models to implementations. This operational semantics is proven equivalent to the denotational semantics used to reason about programs in security proofs. More information about this alternate semantics is provided in Section 4.5, and I show how to reason about implementations in Chapter 7.

3.3 Cryptographic Arguments in FCF

This section contains some examples to describe how cryptographic arguments are completed in FCF. All of the examples in this section are used in proofs in later chapters.

Listing 11 contains the definition of a non-adaptively secure pseudorandom function (PRF). In this definition, the adversary defined by procedures `A1` and `A2` attempts to distinguish two "worlds." In both worlds, the adversary produces a list of values (`lsD`) which are provided to some function, and the corresponding list of outputs (`lsR`) is given back to the adversary. The adversary may also share arbitrary state (`s_A`) between these two procedures. In the first world, the outputs are produced by some function `f`, whereas in the second world these outputs are produced by a random function. This random function is modeled as a stateful oracle called `randomFunc` that keeps track of previous inputs and outputs using a list. The `oracleMap` function is used to map this oracle over the list `lsD`, and `nil` is the initial state of the oracle. The second adversary procedure takes the resulting list of function outputs and the state, and produces a bit. This definition ends by defining the *advantage* of the adversary as the distance between the probability that the adversary produces `true` in these two games. If `f` is a PRF, then this advantage should be "small."

```
Definition PRF_NA_G_A : Comp bool :=
  [lsD, s_A] <-$2 A1;
  lsR <-$ (k <-$ RndKey; ret (map (f k) lsD));
  A2 s_A lsR.

Definition PRF_NA_G_B : Comp bool :=
  [lsD, s_A] <-$2 A1;
  [lsR, _] <-$2 oracleMap randomFunc nil lsD;
  A2 s_A lsR.

Definition PRF_NA_Advantage :=
  | Pr[PRF_NA_G_A] - Pr[PRF_NA_G_B] |.
```

Listing 11: Non-Adaptively Secure Pseudorandom Function

The security definition in Listing 11 can be used as either the end goal of a proof (in order to show that some function is a PRF) or an assumption (to assume that some function is a PRF). I can use this definition as an assumption to unify some game with `PRF_NA_G_A` and another with `PRF_NA_G_B` and replace the distance between these two games with the corresponding `PRF_NA_Advantage`. This technique effectively allows us to rewrite one game with another while adding a "small" value to the bounds produced by the proof.

Listing 12 contains the structure of a hybrid argument[33,32] that bounds the probability that an adversary can distinguish two distributions when given a list of samples from one of the distributions (`ListHybrid_Advantage`). The resulting bound is a function of the advantage of the adversary when attempting to distinguish these two distributions given only a single sample (`DistSingle_Adv`). If the adversary is unlikely to distinguish these distributions when given a single sample, then the adversary is still unlikely to distinguish these distributions when given polynomially many samples. To make this argument more general, the adversary is able to influence the distribution by providing a value (in the case of `DistSingle_G`) or a list of values (in the case of `ListHybrid_G`).

In this listing, `B1` and `B2` (omitted) compose a nat-indexed family of adversaries constructed from `A1` and `A2`, where the *i*th adversary attempts to distinguish the single sample implied by the *i*th distribution in the appropriate hybrid distribution family. In `Single_impl_ListHybrid_sum`, the bound is given as a sum over the advantages of these adversaries, and `maxA` is the maximum size of the list provided by `A1`. If I include an assumption that a single value (`maxAdvantage`) bounds the advantage of each of these adversaries, then we can derive the simpler result of `Single_impl_ListHybrid`.

```
Definition DistSingle_G(c : A -> Comp B) :=
  [a, s_A] <-$2 A1;
  b <-$ c a;
  A2 s_A b.

Definition DistSingle_Adv :=
  | Pr[DistSingle_G c1] - Pr[DistSingle_G c2] |.

Definition ListHybrid_G (c : A -> Comp B) :=
  [lsA, s_A] <-$2 A1;
  lsB <-$ foreach (x in lsA) (c x);
  A2 s_A lsB.

Definition ListHybrid_Adv :=
  | Pr[ListHybrid_G c1] - Pr[ListHybrid_G c2] |.
```

```
Theorem Single_impl_ListHybrid_sum :
  ListHybrid_Adv <=
    sumList (forNats maxA)
      (fun i => DistSingle_Adv c1 c2 (B1 i) B2).

Hypothesis maxAdvantage_correct :
  forall i,
    DistSingle_Adv c1 c2 (B1 i) B2 <= maxAdvantage
         .

Theorem Single_impl_ListHybrid :
  ListHybrid_Adv <= maxA  * maxAdvantage.
```

Listing 12: A Hybrid Argument on Lists

Note that `PRF_NA_Advantage` unifies with `DistSingle_Adv`. So if I assume that some function is a PRF, then I can use the hybrid argument above to conclude that the function is indistinguishable from a random function even when the adversary provides a list of lists of inputs, and receives the result of the PRF mapped over each list (using a different key for each list).

3.4 COMPARISON

In this section, I describe the degree to which FCF and similar systems satisfy the design goals described in Section 3.1. Table 3.1 assigns an informal score to each system for all the design attributes in Section 3.1. For any attribute, a system is scored between 1 and 5, where 1 indicates that the system does not satisfy the goal (or satisfies it poorly), and 5 indicates it satisfies the goal very well. Of course, these scores are intended to be relative, and are only used to compare these systems with each other.

	FCF	EasyCrypt	CertiCrypt	CryptoVerif	F*
Familiarity	4	4	2	4	2
Automation	2	3	2	5	3
Trustworthiness	5	4	5	4	3
Expressivity	4	5	5	2	3
Extensibility	5	3	4	2	3
Concrete Security	5	5	5	5	2
Abstraction	5	4	4	2	2
Implementation	5	4	1	4	4

Table 3.1: Comparison of Mechanized Cryptography Systems

FCF scores well for all attributes except for *Automation*, which is reasonable considering many of the other frameworks were designed to maximize the effect of automation. In the remainder of this section, I will explain the scores in Table 3.1. For each attribute, I will start with the highest-scoring system and then describe the others in comparison.

For *Familiarity*, FCF, EasyCrypt, and CryptoVerif score the highest, and I will describe EasyCrypt first. EasyCrypt is a standalone system, giving the designers complete freedom over the language used to express constructions and security definitions. This language is very natural, and (from personal observation) cryptographers have no trouble understanding definitions in this language. The language of FCF was inspired by the language of EasyCrypt and is similarly familiar, though the language of FCF is influenced by the fact that it is embedded in Coq. Coq's notation system is used extensively by FCF to make definitions more familiar, but a cryptographer reading these definitions will need to learn a few notations in order to understand them. FCF is more familiar than EasyCrypt in its semantics, though. The semantics of FCF assigns a probability distribution to each program using standard set-theoretic notions of probability distributions. In contrast, EasyCrypt is based on a distribution transformer semantics that is much harder for a cryptographer to understand. CryptoVerif is similar to EasyCrypt in that the language is very familiar, but the semantics (based on probabilistic process calculus) is not. A minor issue with CertiCrypt is the fact that reading and understanding security definitions and constructions is somewhat challenging. The core language is similar to that of EasyCrypt, but the deep embedding of this language into Coq requires a large amount of additional syntax to extend the language with new types and operations. The CertiCrypt semantics (which is very similar to the EasyCrypt semantics) is also unfamiliar to cryptographers. Many aspects of F* are unfamiliar to cryptographers, especially the notion of refinement types. The pervasive use of ideal interfaces in F* proofs also forces many cryptographers into an unfamiliar (though easily understandable) style of cryptographic proof.

The system with the highest level of *Automation* is CryptoVerif, which can automatically prove equivalences between intermediate games as well as produce an appropriate sequence of games. But it is important to note that CryptoVerif is not a general-purpose system, and this automation only works due to strict limitations on the types of proof that CryptoVerif is able to consider. EasyCrypt and F* can discharge many goals automatically via their integration with SMT solvers, but the automation in these tools is still very far from the fully-automatic nature of CryptoVerif. The SMT solvers in EasyCrypt are used to solve very simply goals involving logical formulae, but these goals must be produced from a higher-level goal (e.g. the equivalence of two games) manually by using tactics in EasyCrypt. The process is similar in F*, though logical goals are produced by constructing programs in a certain way to give hints to the solver (rather than explicitly applying tactics). EasyCrypt and F* are more general than CryptoVerif, and they notably include looping constructs that are not provided by CryptoVerif. So in order to reason about the behavior of programs in EasyCrypt or F*, it is necessary to determine an appropriate loop invariant or induction hypothesis, which is very hard to do automatically. CertiCrypt and FCF have the lowest level automation because they do not use SMT solvers. Though Coq provides a significant level of proof automation through its tactic language and other features. Through example proofs in Chapters 5 and 6, I demonstrate that the level of automation provided in FCF is sufficient for completing non-trivial proofs with a reasonable about of effort.

CertiCrypt and FCF are the only fully foundational proof frameworks, and therefore they have the most *Trustworthiness*. These frameworks are embedded in Coq, which has a relatively small trusted computing base (TCB) by design, and is used by thousands of people for many different purposes. EasyCrypt and CryptoVerif are standalone tools, and should be considered less trustworthy since they have larger TCBs and fewer users (meaning bugs resulting in unsoundness are less likely to be located). Still it is important to note that the logical frameworks of EasyCrypt and CryptoVerif are simpler than that of Coq, which may increase their trustworthiness in some situations. F* is similar to EasyCrypt and CryptoVerif in that it is a standalone tool with a large TCB. An additional issue with F* is that it cannot perform all of the probabilistic reasoning required to complete a cryptographic proof. So some facts are simply admitted, and it is necessary to inspect these facts in order to trust the proof.

EasyCrypt and CertiCrypt are the most expressive systems. These tools are based on a Turing-complete language that can be used to model any cryptographic scheme or security definition. FCF is similarly expressive, except the language is not Turing-complete. As a result, there may be some cryptographic construction or definition that cannot be modeled precisely in FCF. The language of CryptoVerif has no loops, and the security definitions are limited to secrecy and authenticity. These restrictions severely limit the proofs that can be expressed in CryptoVerif. F* is based on a Turing-complete language which allows the modeling of any cryptographic construction, but the lack of probabilistic reasoning in F* restricts the security definitions and proofs that can be precisely expressed.

FCF was designed to maximize trustworthy *Extensibility*, and it supports the direct incorporation of existing Coq libraries and theory. CertiCrypt can be extended in a way that is equally trustworthy, but the extension suffers from issues related to syntax and familiarity described earlier. EasyCrypt provides a mechanism to add new types and operations along with a set of axioms that describe the behavior of those operations. This mechanism is not trustworthy, however, since these axioms must be inspected in order to ensure that they are reasonable and sound. Also, EasyCrypt cannot be extended with new theory about existing programming language constructs in a trustworthy manner, whereas the theory of FCF and CertiCrypt can be extended by simply proving theorems in Coq. F* can be extended by defining a new type describing the behavior of some operation. Similar to EasyCrypt, it is necessary to inspect these types, and the theory of F* cannot be extended in a trustworthy way. CryptoVerif can be extended to support new types, operations, and security definitions, but these objects must be developed in a particular way so that CryptoVerif's automation can take advantage of them. As a result, extending CryptoVerif is significantly harder compared to the other frameworks.

All systems provide *Concrete Security*, though the claims are significantly weaker in F* because this system is limited in the sorts of probabilistic reasoning it is capable of performing. As a result, a concrete security claim in F* may include an expression describing the behavior of an ideal interface, whereas this expression in other frameworks would be a more precise numerical expression.

FCF takes full advantage of the abstraction mechanisms in Coq to support reusability of definitions, code, and proofs. These mechanisms include higher-order functions, sections, modules, and type classes. CertiCrypt also supports these abstraction mechanisms, though the embedding style of CertiCrypt makes it slightly more difficult to leverage them. EasyCrypt is based on a first order language, but it has a module system that is inspired by the module system of Caml and Coq. This system provides a form of abstraction that is more limited than the systems available in Coq, but it is specifically tailored to problem of developing cryptographic proofs. CryptoVerif and F* are also first order languages, and they provide relatively limited support for reuse through abstraction.

FCF, EasyCrypt, CryptoVerif, and F* have been used to reason about the *Implementation* of cryptographic systems. At present, only FCF has been used to produce a complete, end-to-end proof of security and correctness for a cryptographic implementation. EasyCrypt and CryptoVerif have been used to verify implementations, but the resulting proofs contain small gaps. One of these gaps is that it is necessary to trust that the semantics used to reason about the implementation is compatible with the semantics used to reason about the cryptographic properties of the system. F* is derived from the F# programming language, so reasoning about implementations is very natural, but it is impossible to produce an end-to-end proof of an implementation due to limitations in the cryptographic reasoning ability of F*. CertiCrypt has not been used to reason about implementations of systems, though this is mostly due to the fact that the developers focused their attention on EasyCrypt instead. With some additional effort, CertiCrypt could be just as effective at reasoning about implementations as EasyCrypt or FCF.

3.5 Conclusion

This chapter informally introduced FCF and some criteria against which FCF and similar tools should be evaluated. I also provided a brief assessment of FCF in comparison to other significant cryptographic proof frameworks. Throughout the rest of this paper, I give justification for the assessment of FCF given in Section 3.4. In Chapter 4, I provide a more detailed technical description. Chapter 5 contains several complete example proofs that demonstrate how FCF is used in practice.

4

Technical Description

The previous chapters described cryptographic proofs and gave a brief introduction to developing cryptographic proofs in FCF. Chapter 5 provides several examples of complete proofs in FCF, but first I will describe the technical details of the framework.

FCF provides a common probabilistic programming language (Section 4.1) for describing cryptographic constructions, security definitions, and problems that are assumed to be hard. Then a denotational semantics (Section 4.1) allows reasoning about the probability distributions that correspond to programs in this language. This semantics assigns a numeric value to an event in a probability distribution, and it also allows one to conclude that two distributions are equivalent or are related in other interesting ways.

It can be cumbersome to work directly in the semantics, so FCF provides a theory of distributions (Section 4.2) that can be used to prove that distributions are related by equality, inequality or "closeness." A program logic (Section 4.3) is also provided to ease the development of proofs involving state or looping behavior. As described in Chapter 3, the framework provides a library of tactics and a library of common program elements with associated theory. The equational theory, program logic, tactics, and programming library greatly simplify proof development, yet they are all derived from the semantics of the language, and using them to complete a proof does not reduce the trustworthiness of the proof.

By combining all of the components described above, a developer can produce a proof relating the probability that some adversary defeats the scheme to the probability that some other adversary is able to solve a problem that is assumed to be hard. This is a result in the *concrete setting*, in which probability values are given as expressions, and certain problems are assumed to be hard for particular constructed adversaries. In such a result, it may be necessary to inspect an expression describing a probability value to ensure it is sufficiently "small," or to inspect a procedure to ensure it is in the correct complexity class. FCF provides additional facilities to obtain more traditional asymptotic results, in which these procedures and expressions do not require inspection. A set of asymptotic definitions (Section 4.4) allows conclusions such as "this probability is negligible" or "this procedure executes a polynomial number of queries." In order to apply an assumption about a hard problem, it may be necessary to prove that some procedure is efficient in some sense. So FCF provides an extensible notion of efficiency (Section 4.4.1) and a characterization of non-uniform polynomial time Turing machines.

```
Inductive Comp : Set -> Type :=
| Ret : forall {A : Set}
  {H: EqDec A}, A -> Comp A
| Bind : forall {A B : Set}, Comp B
  -> (B -> Comp A) -> Comp A
| Rnd : forall n, Comp (Bvector n)
| Repeat : forall {A : Set}, Comp A
  -> (A -> bool) -> Comp A.
```

Listing 13: Probabilistic Computation Syntax

$$
\begin{aligned}
[\![\texttt{ret}\ a]\!] &= \mathbf{1}_{\{a\}} \\
[\![x \overset{\$}{\leftarrow} c; f\ x]\!] &= \lambda x. \sum_{b \in \mathrm{supp}([\![c]\!])} \big([\![f\ b]\!]\ x\big)\ \big([\![c]\!]\ b\big) \\
[\![\{0,1\}^n]\!] &= \lambda x.\, 2^n \\
[\![\texttt{Repeat}\ c\ P]\!] &= \lambda x. (\mathbf{1}_P\ x)\ ([\![c]\!]\ x) \\
&\qquad \left(\sum_{b \in P} ([\![c]\!]\ b)\right)^{-1}
\end{aligned}
$$

Figure 4.1: Semantics of Probabilistic Computations

4.1 PROBABILISTIC PROGRAMS

Probabilistic programs are specified using Gallina, the purely functional programming language of Coq, extended with a computational monad in the spirit of Ramsey and Pfeffer[41], that supports drawing uniformly random bit vectors. The syntax of the language is defined by an inductive type called `Comp` and is shown in Listing 13. At a high-level, `Comp` is an embedded domain-specific language that inherits the host language Gallina, and extends it with operations for generating and working with random bits.

The most notable primitive operation is `(Rnd n)`, which produces n uniformly random bits. The `(Repeat c P)` operation repeats a computation `c` until the decidable predicate `P` holds on the value returned. The operations `Bind` and `Ret` are the standard monadic constructors, and allow the construction of sequences of computations, and computations from arbitrary Gallina terms and functions, respectively. However, note that the `Ret` constructor requires a proof of decidable equality for the underlying return type, which is necessary to provide a computational semantics as seen later in this section. In the remainder of this paper, I will use a more natural notation for these constructors: $\{0,1\}^n$ is equivalent to `(Rnd n)`, $x \overset{\$}{\leftarrow} c; f\ x$ is the same as `(Bind c f)`, and `ret e` is `(Ret _ e)`. The framework includes an ASCII form of this notation as seen in the examples in Chapter 3. In the case of `Ret`, the notation serves to hide the proof of decidable equality, which is irrelevant to the programmer and is usually constructed automatically by proof search.

FCF uses a (mostly) *shallow embedding*, in which functions in the object language are realized using functions in the metalanguage. In contrast, CertiCrypt uses a *deep embedding*, in which the data type describing the object language includes constructs for specifying and calling functions, as well as all of the primitives such as bit-vectors and `xor`.

I have found that there are several key benefits to shallow embedding. The primary benefit is that FCF immediately gains all of the capability of the metalanguage, including (in the case of Coq) dependent types, higher-order functions, modules, *etc.* Another benefit is that it is very simple to include any necessary theory in a security proof, and all of the theory that has been developed in the proof assistant can be directly utilized. One benefit that is specific to Coq (and other proof assistants with this property) is that Gallina functions are necessarily terminating, and Coq provides some fairly complex mechanisms for proving that a function terminates. By combining this restriction on functions with additional restrictions on `Repeat`, FCF can ensure that a computation (eventually) terminates, and that this computation corresponds with a distribution in which the total probability mass is 1.

On the other hand, the shallow embedding approach does have some drawbacks. The main drawback is that a Gallina function is opaque; Coq can only reason about a Gallina function based on its input/output behavior. The most significant effect of this limitation is that it is not possible to directly reason about the computational complexity of a Gallina function. This issue is addressed in Section 4.4.1.

The denotational semantics of a probabilistic computation is shown in Figure 4.1. The denotation of a term of type `Comp A` is a function in $A \rightarrow \mathbb{Q}$ which should be interpreted as the probability mass function of a distribution on `A`. In FCF, all distributions are discrete and have finite support. In Figure 4.1, 1_S is the indicator function for set S. So the denotation of `(ret a)` is a function that returns 1 when the argument is definitionally equal to a, and 0 otherwise. We can view the denotation of $x \overset{\$}{\leftarrow} c; f\ x$ as a marginal probability of the joint distribution formed by c and f. We know the probability of all events in c, but we only know the probability of events in f conditioned on events in c, so we can compute the probability of any event in this marginal distribution using the law of total probability. The fact that random bits are uniform and independent is encoded in the denotation of $\{0, 1\}^n$, which is a function that ignores the argument and returns the probability that any n-bit value is equal to a randomly chosen n-bit value. The probability that (`Repeat` c P) produces x is the conditional probability of x given P in c—which is equivalent to the function shown in Figure 4.1.

It is important to note that this language is purely functional, but the monadic style gives programs an imperative appearance. This appearance supports the *Familiarity* design goal since cryptographic definitions and games are typically written in an imperative style.

It is sometimes necessary to include some state in a cryptographic definition or proof. This can be easily accomplished by layering a state monad on top of `Comp`. However, this simple approach does not allow the development of definitions in which an adversary has access to an oracle that must maintain some hidden state across multiple interactions with the adversary. The definition could not simply pass the state to the adversary, because then the adversary could inspect or modify it. So FCF provides an extension to `Comp` for probabilistic procedures with access to a stateful oracle. The syntax of this extended language (Listing 14) is defined in another inductive type called `OracleComp`, where `(OracleComp A B C)` is a procedure that returns a value of type `C`, and has access to an oracle that takes a value of type `A` and returns a value of type `B`.

The `OC_Query` constructor is used to query the oracle, and `OC_Run` is used to run some program under a different oracle that is allowed to access the current oracle. The `OC_Bind` and `OC_Ret` constructors are used for sequencing and for promoting terms into the language, as usual. In the rest

```
Inductive OracleComp :  Set -> Set -> Set -> Type :=
| OC_Query : forall (A B : Set), A -> OracleComp A B B
| OC_Run : forall (A B C A’ B’ S : Set), EqDec S -> EqDec B -> EqDec A ->
    OracleComp A B C -> S -> (S -> A -> OracleComp A’ B’ (B * S)) ->
    OracleComp A’ B’ (C * S)
| OC_Ret : forall A B C, Comp C -> OracleComp A B C
| OC_Bind : forall A B C C’, OracleComp A B C ->
    (C -> OracleComp A B C’) -> OracleComp A B C’.
```

Listing 14: Computation with Oracle Access Syntax

of this paper, I overload the sequencing and `ret` notation in order to use them for `OracleComp` as well as `Comp`. I use `query` and `run`, omitting the additional types and decidable equality proofs, as notation for the corresponding constructors of `OracleComp`.

$$\begin{aligned}
\llbracket \texttt{query}\ a \rrbracket &= \lambda o\ s.(o\ s\ a) \\
\llbracket \texttt{run}\ c'\ s'\ o' \rrbracket &= \lambda o\ s.\llbracket c'(\lambda x\ y.\llbracket (o'(fst\ x)\ y)\ o\ (snd\ x) \rrbracket)\ (s', s) \rrbracket \\
\llbracket \texttt{ret}\ c \rrbracket &= \lambda o\ s.x \overset{\$}{\leftarrow} c; \texttt{ret}\ (x, s) \\
\llbracket x \overset{\$}{\leftarrow} c; f\ x \rrbracket &= \lambda o\ s.[x, s'] \overset{\$}{\leftarrow} \llbracket c\ o\ s \rrbracket; \llbracket (f\ x)\ o\ s' \rrbracket
\end{aligned}$$

Figure 4.2: Semantics of Computations with Oracle Access

The denotation of an `OracleComp` is a function from an oracle and an oracle state to a `Comp` that returns a pair containing the value provided by the `OracleComp` and the final state of the oracle. The type of an oracle that takes an `A` and returns a `B` is `(S -> A -> Comp(B * S))` for some type `S` which holds the state of the oracle. The denotational semantics is shown in Figure 4.2.

4.2 THEORY OF DISTRIBUTIONS

A common goal in a security proof is to compare two distributions with respect to some particular value (or pair of values) in the distributions. To assist with such goals, FCF provides an (in)equational theory for distributions. This theory contains facts that can be used to show that two probability values are equal, that one is less than another, or that the distance between them is bounded by some value. For simplicity of notation, equality is overloaded in the statements below in order to apply to both numeric values and distributions. When I say that two distributions (represented by probability mass functions) are equal, as in $D_1 = D_2$, I mean that the functions are extensionally equal, that is $\forall x, (D_1\ x) = (D_2\ x)$.

Theorem 1 (Monad Laws).

$$\llbracket a \overset{\$}{\leftarrow} \mathtt{ret}\ \ b; f\ a \rrbracket = \llbracket (f\ b) \rrbracket \qquad \llbracket a \overset{\$}{\leftarrow} c; \mathtt{ret}\ \ a \rrbracket = \llbracket c \rrbracket$$

$$\llbracket a \overset{\$}{\leftarrow} (b \overset{\$}{\leftarrow} c_1; c_2\ b); c_3\ a \rrbracket = \llbracket b \overset{\$}{\leftarrow} c_1; a \overset{\$}{\leftarrow} c_2\ b; c_3\ a \rrbracket$$

Theorem 2 (Commutativity).

$$\llbracket a \overset{\$}{\leftarrow} c_1; b \overset{\$}{\leftarrow} c_2; c_3\ a\ b \rrbracket = \llbracket b \overset{\$}{\leftarrow} c_2; a \overset{\$}{\leftarrow} c_1; c_3\ a\ b \rrbracket$$

Theorem 3 (Distribution Irrelevance). For well-formed computation c,

$$\big(\forall x \in supp(\llbracket c \rrbracket), \llbracket f\ x \rrbracket y = v\big) \Rightarrow \llbracket a \overset{\$}{\leftarrow} c; f\ a \rrbracket y = v$$

Theorem 4 (Distribution Isomorphism). For any bijection f,

$$\begin{gathered} \forall x \in supp(\llbracket c_2 \rrbracket), \llbracket c_1 \rrbracket (f\ x) = \llbracket c_2 \rrbracket x \\ \wedge\ \forall x \in supp(\llbracket c_2 \rrbracket), \llbracket f_1\ (f\ x) \rrbracket\ v_1 = \llbracket f_2\ x \rrbracket v_2 \\ \Rightarrow \llbracket a \overset{\$}{\leftarrow} c_1; f_1\ a \rrbracket\ v_1 = \llbracket a \overset{\$}{\leftarrow} c_2; f_2\ a \rrbracket\ v_2 \end{gathered}$$

Theorem 5 (Repeat Equivalence).

$$v1 \in supp(\llbracket c1 \rrbracket) \land P1\ v1 = P2\ v2 = true$$
$$\land\ \llbracket c1 \rrbracket v1 = \llbracket c2 \rrbracket v2 \land \sum_{a \in P1} \llbracket c1 \rrbracket a = \sum_{a \in P2} \llbracket c2 \rrbracket a$$
$$\Rightarrow \llbracket Repeat\ c1\ P1 \rrbracket v1 = \llbracket Repeat\ c2\ P2 \rrbracket v2$$

Theorem 6 (Identical Until Bad).

$$\llbracket a \overset{\$}{\leftarrow} c_1; \texttt{ret}\ (B\ a) \rrbracket = \llbracket a \overset{\$}{\leftarrow} c_2; \texttt{ret}\ (B\ a) \rrbracket \land$$
$$\llbracket a \overset{\$}{\leftarrow} c_1; \texttt{ret}\ (P\ a, B\ a) \rrbracket (x, \mathrm{false}) = \llbracket a \overset{\$}{\leftarrow} c_2; \texttt{ret}\ (P\ a, B\ a) \rrbracket (x, \mathrm{false}) \Rightarrow$$
$$\mathrm{j}\ \llbracket a \overset{\$}{\leftarrow} c_1; \texttt{ret}\ (P\ a) \rrbracket\ x\ \ \llbracket a \overset{\$}{\leftarrow} c_2; \texttt{ret}\ (P\ a) \rrbracket\ x\, \mathrm{j} \leq \llbracket a \overset{\$}{\leftarrow} c_1; \texttt{ret}\ (B\ a) \rrbracket\ \mathrm{true}$$

The meaning and utility of many of the above theorems is direct (such as the standard monad properties in Theorem 1), but others require some explanation. Theorem 3 considers a situation in which the probability of some event y in $\llbracket f\ \ x \rrbracket$ is the same for all x produced by computation c. Then the distribution $\llbracket c \rrbracket$ is irrelevant, and it can be ignored. This theorem only applies to *well-formed* computations: A well-formed computation is one that terminates with probability 1, and therefore corresponds to a valid probability distribution.

Theorem 4 is a powerful theorem that corresponds to the common informal argument that two random variables "have the same distribution." More formally, assume distributions $\llbracket c_1 \rrbracket$ and $\llbracket c_2 \rrbracket$ assign equal probability to any pair of events $(f\ \ x)$ and x for some bijection f. Then a pair of sequences beginning with c_1 and c_2 are denotationally equivalent as long as the second computations in the sequences are equivalent when conditioned on $(f\ \ x)$ and x. A special case of this theorem is when f is the identity function, which allows us to simply "skip" over two semantically equivalent computations at the beginning of a sequence.

Theorem 5 is a simple rule that can be used to show a form of equivalence between a pair of `Repeat` statements. This theorem assumes that the underlying computations are equivalent w.r.t. a pair of values $v1$ and $v2$, and the events that cause the `Repeat` statements to terminate have the same probability mass. Then the theorem states that the repeat statements are equivalent w.r.t. the pair of values $v1$ and $v2$.

Theorem 6, also known as the "Fundamental Lemma"[15], is typically used to bound the distance between two games by the probability of some unlikely event. Computations c_1 and c_2 produce both a value of interest and an indication of whether some "bad" event happened. We use (decidable) predicate $\boldsymbol{B}$ to extract whether the bad event occurred, and projection $\boldsymbol{P}$ to extract the value of interest. If the probability of the "bad" event occurring in c_1 and c_2 is the same, and if the distribution of the value of interest is the same in c_1 and c_2 when the bad event does not happen, then the distance between the probability of the value of interest in c_1 and and c_2 is at most the probability of the "bad" event occurring.

4.3 Program Logic

The final goal of a cryptographic proof is always some relation on probability distributions, and in some cases it is possible to complete the proof entirely within the equational theory described in 4.2. However, when the proof requires reasoning about loops or state, a more expressive theory may be needed in order to discharge some intermediate goals. For this reason, FCF includes a program logic that can be used to reason about changes to program state as the program executes. Importantly, the program logic is related to the theory of probability distributions through completeness and soundness theorems which allow the developer to derive facts about distributions from program logic facts, and vice-versa.

The core logic is a Probabilistic Relational Postcondition Logic (PRPL), that behaves like a

Hoare logic, except there are no preconditions. The definition of a PRPL specification is given in Definition 1. In less formal terms, computations p and q are related by the predicate Φ if both p and q are marginals of the same joint probability distribution, and Φ holds on all values in the support of that joint distribution.

Definition 1 (PRPL Specification). Given p : `Comp A` and q : `Comp B`,

$$p \sim q\{\Phi\} \Leftrightarrow \begin{pmatrix} \exists\,(d : \texttt{Comp (A * B)}), \forall (x, y) \in \mathrm{supp}([\![d]\!]), \Phi\ x\ y\ \wedge \\ [\![p]\!] = [\![x \overset{\$}{\leftarrow} d; \texttt{ret}\ (\mathrm{fst}\ x)]\!] \wedge [\![q]\!] = [\![x \overset{\$}{\leftarrow} d; \texttt{ret}\ (\mathrm{snd}\ x)]\!] \end{pmatrix}$$

Using the PRPL, it is possible to construct a Probabilistic Relational Hoare Logic (PRHL) which includes a notion of precondition for functions that return computations as shown in Definition 2. The resulting program logic is very similar to the Probabilistic Relational Hoare Logic of EasyCrypt[9], and it has many of the same properties.

Definition 2 (PRHL Specification). Given p : `A -> Comp B` and q : `C -> Comp D`, $\{\Psi\}p \sim q\{\Phi\} \Leftrightarrow \forall a\ b, \Psi\ a\ b \Rightarrow (p\ a) \sim (q\ b)\{\Phi\}$.

Several theorems are provided along with the program logic definitions to simplify reasoning about programs. In order to use the program logic, one only needs to apply the appropriate theorem, so it is not necessary to produce the joint distribution described in the definition of a PRPL specification unless a suitable theorem is not provided. Theorems are provided for reasoning about the basic programming language constructs, interactions between programs and oracles, specifications describing equivalence, and the relationship between the program logic and the theory of probability distributions. Some of the more interesting program logic theorems are described below.

Theorem 7 (Soundness/Completeness).

$$p \sim q\{\lambda\ a\ b.a = x \Leftrightarrow b = y\} \Leftrightarrow [\![p]\!]\ x = [\![q]\!]\ y$$

$$p \sim q\{\lambda\ a\ b.a = x \Rightarrow b = y\} \Leftrightarrow [\![p]\!]\ x \leq [\![q]\!]\ y$$

Theorem 8 (Sequence Rule).

$$p \sim q\{\Phi'\} \Rightarrow \{\Phi'\}r \sim s\{\Phi\} \Rightarrow (x \overset{\$}{\leftarrow} p; r\ x) \sim (x \overset{\$}{\leftarrow} q; s\ x)\{\Phi\}$$

Theorem 9 (Oracle Equivalence). Given an `OracleComp` c, and a pair of oracles, o and p with initial states s and t,

$$\Phi = \lambda\ x\ y.(\text{fst}\ x) = (\text{fst}\ y) \wedge P\ (\text{snd}\ x)(\text{snd}\ y) \Rightarrow$$

$$\big(\forall a\ s'\ t', P\ s'\ t' \Rightarrow (o\ s'\ a) \sim (p\ t'\ a)\{\Phi\}\big) \Rightarrow P\ s\ t \Rightarrow ([\![c]\!]\ o\ s) \sim ([\![c]\!]\ p\ t)\{\Phi\}$$

Theorem 7 relates judgments in the program logic to relations on probability distributions. Theorem 8 is the relational form of the standard Hoare logic sequence rule, and it supports the decomposition of program logic judgments. Theorem 9 allows the developer to replace some oracle with an observationally equivalent oracle. There is also a more general form of this theorem (omitted for brevity) in which the state of the oracle is allowed to go bad. This more general theorem can be combined with Theorem 6 to get "identical until bad" results for program/oracle interactions.

4.4 Asymptotic Theory

Using the tools described in the previous sections, it is possible to complete a proof of security in the *concrete setting*. That is, the probability that an adversary wins a game is given as an expression which may include some value (or set of values) η that we can interpret as the security parameter. To get a typical asymptotic security result, I must show that this expression, when viewed as a function of η, is negligible. To assist with these sorts of conclusions, FCF provides a library of asymptotic definitions such as Definitions 3 and 4. The library also includes theorems that can be used to prove that functions are polynomial or negligible based on their composition(*e.g.*, the sum of polynomials is polynomial, the quotient of polynomial and exponential is negligible).

Definition 3 (At Most Polynomial). A function $f : \mathbb{N} \to \mathbb{N}$ is *at most polynomial* iff $\exists x, c_1, c_2, \forall n, f(n) \leq c_1 \; n^x + c_2$

Definition 4 (Negligible Function). A function $f : \mathbb{N} \to \mathbb{Q}$ is *negligible* iff $\forall c, \exists n, \forall x > n, f(x) < {}^1\!/_{x^c}$

4.4.1 Efficient Procedures

A typical asymptotic security property states that a family of cryptographic schemes has some desirable property for all efficient adversaries. So in order to prove and apply these properties, we require some notion of "efficient" (families of) procedures. The language of computations used in FCF does not imply any particular model of computation—it is just a mechanism to specify probability distributions in a computational manner. Any notion of "efficiency" must first fix a model of computation, and then a complexity class on that model. This notion of efficiency should be flexible and extensible so FCF can support several different models of computation and complexity classes.

To accomplish this flexibility, asymptotic security definitions are parameterized by an "admissibility predicate" indicating the class of adversaries against which a problem is assumed to be hard, or a scheme is proven to be secure. In this setting, the adversary is a family of procedures indexed by a natural number which indicates the value of the security parameter. The admissibility predicate can describe the efficiency of the adversary as well as other properties such as well-formedness or the number of allowed oracle queries as a function of the security parameter.

FCF includes a simple cost model and an associated admissibility predicate describing non-uniform worst-case polynomial time Turing machines that perform a (worst case) polynomial number of oracle queries. This admissibility predicate is constructed using a concrete cost model that assigns numeric costs to particular Coq functions, `Comp` values, and `OracleComp` values. In this cost model, the cost of executing a function is in $\mathbb{N}$, indicating the worst-case (over all arguments) execution time. The cost of running a `Comp` is in $\mathbb{N}$, indicating the worst-case execution time over all outcomes. The cost of executing an `OracleComp` is in $\mathbb{N} \rightarrow \mathbb{N}$, and is a function from the cost of executing the oracle to the cost of executing the computation, including the cost of executing all oracle queries.

The cost model for Gallina functions is axiomatic, as there is no direct way to capture such an intensional property for these terms. The cost model includes axioms for primitive operations as well as a set of combinators for building more complicated functions. For example, the model includes an axiom stating that the `xor` operation for bit vectors of length c has a cost of c. As other examples, the model includes axioms stating that the cost of f composed with g is the sum of the costs of f and g, and the cost of $(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3)$ is the cost of e_1 plus the maximum of the costs of e_2 and e_3.

The axiomatic nature of the cost model allows it to be easily extended – if a proof uses a function that is not defined in this cost model, the proof can assume an axiom describing the cost of the function. Obviously, these cost axioms are incomplete, but in practice, the number required is relatively small since it is only necessary to reason about the cost of functions used by a constructed adversary in a proof. Of course, the axioms need to be carefully inspected to ensure they accurately describe the desired complexity class, though a similar kind of inspection is needed to ensure the faithfulness of a cost model for a deeply-embedded language.

It is also important to note that the efficiency of a constructed adversary in FCF is established in an *extensional* manner. That is, by showing that some procedure is associated with a particular cost, I am proving an upper bound on the minimum cost over all equivalent procedures. This result is sufficient for a reduction, since the obligation is to show the existence of an efficient procedure. Also, a proof that a Gallina term has some particular complexity does not imply that any extracted OCaml code will have this complexity.

4.5 Operational Semantics and Reasoning about Code

FCF includes a mechanism for reasoning about implementations that provides a strong guarantee of equivalence between a model of a probabilistic program and the code implementing the model. The framework includes a small-step operational semantics (Figure 4.3) that describes the behavior of FCF computations on a traditional machine (in which the memory contains values rather than probability distributions). This operational semantics is an oracle machine that is given a finite list of bits representing the "random" input, and it describes how a computation takes a single step to produce a new computation(*more*), a final value(*done*), or fails due to insufficient input bits(*eof*).

In the operational semantics, the "random" inputs are provided in the list of bits s. When random inputs are requested, these bits are shifted out of the list and given to the program, and the rest of the list becomes the new value of s. Note that I chose to model the random input as a list instead of a stream in order to simplify the development in Coq, and also to allow reasoning about systems that are only given finite "random" input.

There is only one rule for *ret* in this semantics, and this rule passes along s untouched and states that the computation is complete and the final value is the value that was supplied to the *ret* constructor. There are three possible ways for a sequence to take a step, depending on what happens when the first computation in the sequence takes a step. In essence, the first computation is executed until it is done, and then the resulting value is given to the function defining the second computation. If the random bits are exhausted when the first computation is running, then the entire

$$\frac{}{ret\ a, s \to done\ a, s}$$

$$\frac{(c, s) \to done\ a, s'}{(x \overset{\$}{\leftarrow} c;\ f\ \ x), s \to more\ (f\ \ a),\ s'}$$

$$\frac{(c, s) \to more\ c', s'}{(x \overset{\$}{\leftarrow} c;\ f\ \ x), s \to more\ (x \overset{\$}{\leftarrow} c';\ f\ \ x), s'}$$

$$\frac{(c, s) \to eof}{(x \overset{\$}{\leftarrow} c;\ f\ \ x), s \to eof}$$

$$\frac{shiftOut\ s\ n = Some\ (v, s')}{\{0, 1\}^n, s \to more\ (ret\ v)\ s'}$$

$$\frac{shiftOut\ s\ n = None}{\{0, 1\}^n, s \to eof}$$

$$\frac{}{Repeat\ c\ P, s \to more\ (x \overset{\$}{\leftarrow} c;\ if\ (P\ x)\ then\ (ret\ x)\ else\ (Repeat\ c\ P)),\ s}$$

Figure 4.3: Small-step Operational Semantics

sequence fails to complete due to bit exhaustion. The sampling operation simply steps to *(ret v)* when *v* can be shifted out of the list, or *eof* if there are insufficient bits. The *Repeat* operation takes one step to the appropriate sequence that runs the underlying computation, tests for the termination condition, and performs another *Repeat* if the termination condition is not met.

To show that this semantics is correct, I consider $[c]_n$, the multiset of results obtained by running a program c under this semantics on the set of all input lists of length n. One can interpret $[c]_n$ as a distribution where the mass of some value a in the distribution is the proportion of input strings that cause the program to terminate with value a. The statement of equivalence between the semantics is shown in Theorem 10.

Theorem 10. If c is well-formed, then $\lim_{n \to \infty} [c]_n = [\![c]\!]$

FCF contains a proof of Theorem 10 as a validation of the operational semantics used for extraction and reasoning about implementations. This proof is described in Appendix A.

To obtain an implementation from a model, one can use the standard Coq extraction mechanism to extract the operational semantics along with the model of interest and all supporting types and functions. This semantics can also be used to prove that an implementation in C (or any language that can be modeled in Coq) is equivalent to the model and therefore shares some of its security properties. Both of these techniques for producing verified implementations are described in Chapter 7.

This alternate semantics also provides other benefits. Because limits are unique, if two programs are equivalent under the operational semantics, then they are also equivalent under the denotational semantics. This allows us to prove equivalence of two programs using the operational semantics when it is more convenient to do so. Another benefit is that the operational semantics can be considered to be the basic semantics for computations, and the denotational semantics no longer needs to be trusted. Some may prefer this arrangement, since the operational semantics more closely resembles a typical model of computation, and may be easier to understand and inspect. The operational semantics can also be used as a basis for a model of computation used to determine whether programs are efficient.

4.6 Related Work

There has been a large amount of work in the area of verifying cryptographic schemes in recent years. In this section we will describe some of this related work, focusing on systems that attempt to establish security in the computational model. CertiCrypt[11] and EasyCrypt[9] have been thoroughly discussed previously in this paper.

There are several other examples of frameworks for cryptographic security proofs implemented within proof assistants. The most similar work is that of Nowak[40], who was the first to develop proofs of cryptography in Coq using a shallow embedding in which programs have probability distributions as their denotations. FCF builds on this work by adding more tools for modeling and reasoning such as procedures with oracle access (Section 4.1), a program logic (Section 4.3), and asymptotic reasoning (Section 4.4).

The work of Affeldt et al.[2] is a Coq library utilizing a deeply-embedded imperative programming language. This library is a predecessor to CertiCrypt, and it includes some important elements that were later adopted by CertiCrypt. Notably, the probabilistic programming language in this work is given a semantics in which program states are distributions, and the semantics describes how these distributions are transformed by each command in the language. CertiCrypt and EasyCrypt extended this work by adding language constructs such as oracles and unrestricted loops, and well as reasoning tools such as the Probabilistic Relational Hoare Logic.

Verypto[17] is a fully-featured framework built on Isabelle[39] that includes a deep embedding of a functional programming language. To allow state information to remain hidden from adversaries, Verypto provides ML-style references, in contrast to the oracle system provided by FCF. To date, Verypto has only been used to prove the security of simple constructions, but this work uses an interesting approach that deserves more exploration.

CryptoVerif[20] is a tool based on a concurrent, probabilistic process calculus that is only able to prove properties related to secrecy and authenticity. CryptoVerif is highly automated to the extent that it will even attempt to locate intermediate games, and so proof development in CryptoVerif requires far less effort compared to FCF or EasyCrypt. However, there are a large number of proofs that could be completed in FCF or EasyCrypt that are impossible in CryptoVerif due to its specialized nature.

Refinement types[16] have been used by Fournet et al[31] to develop proofs of security for cryptographic schemes in the computational model. In this system, a security property is specified as an ideal functionality (in the sense of the real/ideal paradigm), and proofs are completed using the "sequence of games" style. This system is limited by the fact that the language is not probabilistic, and it must simply be assumed that the behavior of the ideal functionality is similar to the corresponding real functionality. This approach allows the proofs of security to be fairly simple, but no concrete security claims are proved, so it may be difficult to make practical claims based on such a proof.

Computational soundness[1] provides another mechanism for verifying cryptographic schemes. This approach attempts to derive security in the computational model from security in the symbolic model by showing that any likely execution trace in the computational model also exists in the symbolic model. It is possible to mechanize such a proof as described in[6]. This approach is limited to classes of schemes for which computational soundness results have been discovered. Another limitation with this approach is that it can only produce proofs in the asymptotic setting—there is no way to prove concrete security claims.

Protocol Composition Logic (PCL)[27] provides a logic and proof system for verifying cryptographic schemes in the symbolic model. The system is based on a process calculus and allows reasoning about the results of individual protocol steps. More recent work[28] has extended this logic to allow for proofs in the computational model. In computational PCL, formulas are interpreted against probability distributions on traces and a formula is true if it holds with overwhelming probability. This approach is similar to computational soundness in that low-probability traces are ignored, and proofs of concrete security claims are impossible.

4.7 CONCLUSION

FCF is designed in such a way that the language semantics is simple and easy to understand. Using this semantics as a foundation, I build a sophisticated set of tools for reasoning about cryptographic systems. These tools, including a theory of distributions, a program logic, and a library of programming constructions, are proved correct within Coq. The resulting system can be used to develop and check cryptographic proofs without trusting any more than the semantics of the language and the Coq proof checker.

I show in Chapter 5 and Chapter 6 how to complete proofs in this framework. Appendix A contains more technical details on the operational semantics and the proof that relates the operational semantics to the denotational semantics.

5

Example Proofs

Chapter 3 included some simple examples in order to introduce FCF and its components. In this chapter, I describe several complete cryptographic proofs in order to explain proof development in FCF and illustrate several aspects of the framework. The examples in this chapter are relatively simple, and they include proofs of security for encryption schemes and pseudorandom generators. Chapter 6 contains a proof of a complex searchable symmetric encryption scheme that demonstrates the scalability of FCF. Chapter 7 includes a description of a proof of security for HMAC that is used to show that an implementation of this construction is secure.

5.1 El Gamal Encryption

I begin with a mechanized proof of security for El Gamal[30] encryption. This proof is relatively simple, and many of the details of the proof are provided for illustration purposes. Later proofs will omit some details for the sake of brevity.

```
Class Group := {
  GroupElement : Set;
  groupOp :
    GroupElement -> GroupElement -> GroupElement;
  identity : GroupElement;
  inverse : GroupElement -> GroupElement;

  associativity : forall (x y z : GroupElement),
    groupOp (groupOp x y) z =
    groupOp x (groupOp y z);

  left_identity : forall (a : GroupElement),
    groupOp identity a = a;

  right_identity : forall (a : GroupElement),
    groupOp a identity = a;

  left_inverse : forall (a : GroupElement),
    groupOp (inverse a) a = identity;

  right_inverse : forall (a : GroupElement),
    groupOp a (inverse a) = identity

}.

(* Introduce a new scope. *)
Section GroupProperties.

   (* Assume we have a Group in this scope. *)
  Context ‘{G : Group}.

  (* Define exponentiation for group elements *)
  Fixpoint groupExp(a : GroupElement)(n : nat) :=
    match n with
      | 0 => identity
      | S n’ => groupOp a (groupExp a n’)
    end.

  Lemma groupExp_identity : forall n,
    groupExp identity n = identity.
        ..
  Qed.

  Theorem groupExp_plus : forall n1 n2 x,
    groupExp x (n1 + n2) =
    groupOp (groupExp x n1) (groupExp x n2).
        ..
  Qed.

  Theorem groupExp_mult : forall n2 n1 x,
    (groupExp (groupExp x n1) n2) =
    (groupExp x (n1 * n2)).
        ..
  Qed.

End GroupProperties.
```

Listing 15: Group Definition and Facts

5.1.1 Cyclic Groups

El Gamal encryption is based on the assumed hardness of certain problems related to cyclic groups. FCF includes a definition of groups and finite cyclic groups (Listings 15 and 16), as well as a set of

```
Class FiniteCyclicGroup(G : Group) := {
  generator : GroupElement;
  order : posnat;
  groupLog : GroupElement -> nat;
  group_cyclic: forall (a : GroupElement),
    generator^(groupLog a) = a;
  groupLog_correct: forall x,
    modNat (groupLog (generator^x)) order =
    modNat x order;
  groupIdent : generator^0 = identity;
  groupOrder : generator^order = generator^0

}.

Section FiniteCyclicGroupProperties.

  Context`{FCG : FiniteCyclicGroup}.

  Theorem groupExp_eq : forall x y,
    modNat x order = modNat y order <->
    generator^x = generator^y.
        ...
  Qed.

  Theorem commutativity : forall x y,
    x * y = y * x.
        ...
  Qed.

  Theorem groupExp_distrib : forall n x y,
    (x * y)^n = x^n * y^n.
        ...
  Qed.

  Theorem ident_l_unique : forall x y,
    x * y = y ->
    x = identity.
        ...
  Qed.

  Theorem groupExp_mod : forall n,
    generator^n = generator^(modNat n order).
        ...
  Qed.

End FiniteCyclicGroupProperties.
```

Listing 16: Finite Cyclic Group Definition and Facts

facts about these objects that are proven from the assumptions in the definitions. I use Coq's notation system to assign infix `*` to mean `groupOp` and infix ˆ to mean `groupExp`. The type class mechanism of Coq allows these definitions and facts to be easily incorporated into a security proof.

5.1.2 El Gamal Encryption

The El Gamal key generation, encryption, and decryption algorithms are provided in Listing 17. In this listing, the `[0 .. order)` notation invokes the `RndNat` construction introduced in Section 3.2.2 to produce a uniform natural number that is less than the order of the group. I can prove that the decryption algorithm is correct as shown in Listing 18. In this theorem `getSupport` is a function that returns the support of the distribution corresponding to the specified computation. This theorem considers any key pair that is produced by the key generation routine and any message and ciphertext that is produced by encrypting that message. The theorem states that decrypting the ciphertext using the appropriate key produces the original message.

The proof of correctness of the decryption function begins by unfolding all the relevant definitions. Then Coq's `intuition` tactic introduces all of the required hypotheses. The `simp_in_support` tactic, which is provided by FCF, is an automated tactic that locates hypotheses stating that some value is in the support of some distribution and replaces these hypotheses with more informative ones. For example if I have that `x` is in the support of `a <-$ c1; (c2 a)`, then `simp_in_support` will replace this hypothesis with a new variable `y` and assumptions that `y` is in the support of `c1` and `x` is in the support of `(c2 y)`. This tactic performs substitution and other simplifications as well, and following the application of this tactic I can complete the proof by rewriting and applying some assumptions and results from group theory and arithmetic.

```
Definition ElGamalKeygen :=
  m <-$ [0 .. order);
  ret (m, g^m).

Definition ElGamalEncrypt(msg key : GroupElement) :=
  m <-$ [0 .. order);
  ret (g^m, key^m * msg).

Definition ElGamalDecrypt(key : nat)(ct : GroupElement * GroupElement) :=
  [c1, c2] <-2 ct;
  s <- c1^key;
  (inverse s) * c2.
```

Listing 17: El Gamal Encryption

```
Theorem ElGamalDecrypt_correct :
  forall (pubkey msg : GroupElement)(prikey : nat)(ct : GroupElement * GroupElement),
    In (prikey, pubkey) (getSupport ElGamalKeygen) ->
    In ct (getSupport (ElGamalEncrypt msg pubkey)) ->
    ElGamalDecrypt prikey ct = msg.

  unfold ElGamalKeygen, ElGamalEncrypt, ElGamalDecrypt.
  intuition. repeat simp_in_support.
  rewrite <- associativity.
  repeat rewrite groupExp_mult.
  rewrite mult_comm.
  rewrite left_inverse.
  apply left_identity.
Qed.
```

Listing 18: El Gamal Key Decryption Correctness

5.1.3 The Decisional Diffie Hellman Problem

El Gamal derives its security from the assumed hardness of the Decisional Diffie Hellman Problem, described in Listing 19. The definitions for this problem are parameterized on an abstract procedure `A`. Intuitively, `A` is an adversary which finds itself in one of two “worlds”, `DDH0` or `DDH1`. At the end of the procedure, `A` outputs a bit in order to indicate the world in which it believes it resides. According to the DDH assumption, if `A` is computationally efficient (e.g. probabilistic polynomial time), then it can only distinguish these two worlds with negligible probability.

```
Section DDH.

  Context‘{FCG : FiniteCyclicGroup}.
  Variable A : (GroupElement * GroupElement * GroupElement) -> Comp bool.

  Definition DDH0 :=
    x <-$ [0 .. order);
    y <-$ [0 .. order);
    b <-$ (A (g^x, g^y, g^(x * y)));
    ret b.

  Definition DDH1  :=
    x <-$ [0 .. order);
    y <-$ [0 .. order);
    z <-$ [0 .. order);
    b <-$ (A (g^x, g^y, g^z));
    ret b.

  Definition DDH_Advantage:= | Pr[DDH0] - Pr[DDH1] |.

End DDH.
```

Listing 19: Decisional Diffie Hellman

5.1.4 Indistinguishability under Chosen Plaintext Attack

I will show that El Gamal ciphertexts are indistinguishable under chosen plaintext attack (IND-CPA) as defined in Listing 20. The definition of IND-CPA is parameterized on an abstract key generation procedure (Gen), encryption procedure (Enc), and adversary procedures (A1 and A2). I can conclude that some encryption scheme (G, E) is secure in the sense of IND-CPA if $Adv_{\text{IND-CPA}}(G, E, A1, A2)$

is small for all A1 and A2. Intuitively, this means that the adversary composed of A1 and A2 cannot efficiently distinguish the encryptions of any two plaintexts that it is capable of efficiently producing.

Note that the definition of IND-CPA allows the two adversary procedures to share state, which is performed by receiving a state object from the first procedure and giving it to the second procedure. Proofs of security using this definition will be quantified over all adversary procedures and all types of state.

```
Section IND_CPA.

  Variable Plaintext : Set.
  Variable Ciphertext : Set.
  Variable PrivateKey : Set.
  Variable PublicKey : Set.

  Variable KeyGen : Comp (PrivateKey * PublicKey).
  Variable Encrypt : Plaintext -> PublicKey -> Comp Ciphertext.

  Variable A_state : Set.
  Variable A1 : PublicKey -> Comp (Plaintext * Plaintext * A_state).
  Variable A2 : (Ciphertext * A_state) -> Comp bool.

  Definition IND_CPA_G :=
    [prikey, pubkey] <-$2 KeyGen;
    [p0, p1, a_state] <-$3 (A1 pubkey);
    b <-$ {0, 1};
    pb <- if b then p0 else p1;
                         c <-$ (Encrypt pb pubkey);
                         b’ <-$ (A2 (c, a_state));
                         ret (eqb b b’).

  Definition IND_CPA_Advantage := | Pr[IND_CPA_G] - 1 / 2 |.

End IND_CPA.
```

Listing 20: Indistinguishability under Chosen Plaintext Attack

5.1.5 Proof of Security

A typical approach to proving the security of El Gamal encryption is to show that Theorem 11 is true, thus contradicting our assumption that the DDH problem is hard. I will actually prove Theorem 12, which is a stronger theorem, and which isolates the equivalence goal from the efficiency goal, allowing me to prove them independently.

Theorem 11. For all efficient A1 and A2 for which

$$Adv_{\text{IND-CPA}}(\mathit{ElGamalGen}, \mathit{ElGamalEnc}, A1, A2)$$

is non-negligible, there exists efficient B such that $Adv_{DDH}(B)$ is non-negligible.

Theorem 12. For all A1 and A1, there exists B such that B is efficient if A1 and A2 are efficient, and

$$Adv_{\text{IND-CPA}}(\mathit{ElGamalGen}, \mathit{ElGamalEnc}, A1, A2) = \\ Adv_{DDH}(B)$$

```
Definition B(gx gy gz : GroupElement) : Comp bool :=
    [s, p0, p1] <--** A1(gx);
    b <- {0,1};
    pb <-! if b then p0 else p1;
    c <-! (gy, gz * pb);
    b’ <- (A2 s c);
    ret (eqb b b’).

Theorem ElGamal_IND_CPA_Advantage :
  IND_CPA_Advantage ElGamalKeygen ElGamalEncrypt A1 A2 ==
  DDH_Advantage B.
```

Listing 21: DDH Distinguisher

I will use the procedure defined in Listing 21 as the witness to prove Theorem 12. First, it is obvious that B can be constructed from any A1 and A2. For simplicity, I do not formally prove that B is efficient (assuming A1 and A2 are efficient), but this fact can be established by inspection. This listing also contains the statement of the main theorem in Coq notation. This statement is an equality on distances, and I prove this by showing that the corresponding terms in the distance are equal, and thus the distances must be equal.

Listing 22 contains the statement of equality on the first pair of terms along with the proof of this fact. Each line of the proof contains the application of a single tactic. Most of these tactics simply inline definitions and swap the order of statements in order to get identical statements at the beginning of the procedures. Once the procedures begin with identical statements, they can be removed using comp_skip. I rewrite with the groupExp_mult identity (from the group theory library) toward the end of the proof in order to justify that the statements at the beginning of the procedures are identical. I use `intuition` to discharge trivial goals, such as establishing the equality of two terms that are syntactically identical. Note that `dist_at` is a *tactical* (a higher-order tactic) that accepts a tactic and a location (left computation or right computation and statement number) at which the tactic should be applied. This tactical is used in this proof to inline statements that are not at the beginning of a computation.

```
Theorem ElGamal_IND_CPA0 :
  Pr[IND_CPA_G ElGamalKeygen ElGamalEncrypt A1 A2] ==
  Pr[DDH0 B].

  unfold IND_CPA_G, DDH0, ElGamalKeygen, ElGamalEncrypt, B.

  inline_first.
  comp_skip.

  dist_at dist_inline rightc 1.
  comp_swap rightc.
  comp_skip.

  destruct x0.
  destruct p.

  dist_at dist_inline rightc 1.
  comp_swap rightc.
  comp_skip.

  comp_inline leftc.
  comp_skip.

  comp_inline rightc.
  comp_skip.
  rewrite groupExp_mult; intuition.

  comp_simp.
  intuition.
Qed.
```

Listing 22: Proof of Equality of First Terms

The proof of equality for the remaining terms is easier if I introduce some intermediate games and prove the equality in several steps. Procedures G1 and G2 (Listing 23) are used to prove that

$\Pr[DDH1(B) = 1] = 1/2$ one step at a time by transitivity of equality. These procedures use a subroutine called RndG that uniformly samples an element from the group.

```
Definition G1 :=
  gx <-$ RndG;
  gy <-$ RndG;
  [p0, p1, s] <-$3 (A1 gx);
  b <-$ {0, 1};
  gz' <-$ (
  pb <- if b then p0 else p1;
  gz <-$ RndG ; ret (gz * pb));
  b' <-$ (A2 (gy, gz', s));
  ret (eqb b b').
```

```
Definition G2 :=
  gx <-$ RndG;
  gy <-$ RndG;
  [p0, p1, s] <-$3 (A1 gx);
  gz <-$ RndG ;
  b' <-$ (A2 (gy, gz, s));
  b <-$ {0, 1};
  ret (eqb b b').
```

Listing 23: ElGamal Proof Intermediate Procedures

These procedures are related to the `DDH1` game and to each other by equality as shown in Listing 24. The proofs of these facts are omitted, but summarized here. The first fact follows only from reordering of independent statements by Theorem 2 (Commutativity). The second proof is essentially a one-time pad argument which is summarized here. The primary difference between procedures `G1` and `G2` is that the second parameter given to A2 is a random group element in G2, but in G1 it is the product of a random group element and a particular group element. This is a form of one-time pad, so I can show that these values are equivalent. This argument is formalized in the one-time pad (OTP) module that is included in the FCF library. In order to apply this argument, I instantiate the "adversary" in the one-time pad proof using the remaining computation of G1 and G2 (after the one-time pad is applied).

```
Theorem ElGamal_G1_DDH1 :
  Pr [G1] == Pr [DDH1 B].

Theorem ElGamal_G1_G2 :
  Pr[G1] == Pr[G2].
```

Listing 24: Equivalence of Intermediate Procedures

The last fact that I need is that the probability that the adversary produces `true` in game `G2` is exactly one half. This proof can be completed by removing all of the statements before the coin flip using the distribution irrelevance theorem (Theorem 3), and then invoking the automated `dist_compute` tactic to compute this probability value. Given this theorem, we can apply transitivity of equality to show that the probability that the adversary produces `true` in game `DDH1` is one-half. These theorems are stated in Listing 25.

```
Theorem ElGamal_G2_OneHalf :
  Pr [G2] == 1 / 2.

Theorem ElGamal_DDH1_OneHalf :
  Pr [DDH1 B] == 1 / 2.
```

Listing 25: Calculated Probability of Game G2

At this point, I have all the facts necessary to prove the theorem stated in Listing 21. The theorem in Listing 22 establishes the equality of the first pair of terms, and the final result of Listing 25 establishes the equality of the second pair of terms. Thus the distances are equal.

5.2 Symmetric Encryption from a Pseudorandom Function

The next example considers a simple encryption scheme constructed from a pseudorandom function (PRF), and I prove that ciphertexts produced by this scheme are IND-CPA. This example proof is only slightly more complex than the El Gamal example (Section 5.1.2), and yet it contains many of the elements that one would find in a typical cryptographic proof. As a result, this example exercises all of the key functionality of FCF. Notably, this proof gives a result in the concrete setting and then uses that result to develop an asymptotic security claim.

5.2.1 Concrete Security Definitions

In FCF, concrete security definitions are used to describe properties that some construction is proven to have, as well as problems that are assumed to be hard. In the PRF encryption proof, I use the definition of a PRF to assume that such a PRF exists, and I use that assumption to prove that the construction in question has the IND-CPA property. A concrete security definition typically contains

```
Variable Key D R : Set.
Variable RndKey : Comp Key.
Variable RndR : Comp R.
Variable A : OracleComp D R bool.
Variable f : Key -> D -> R.

Definition f_oracle(k: Key)(x: unit)
  (d : D) : Comp (R * unit) :=
  ret (f k d, tt).
Definition PRF_G_A : Comp bool :=
  k <-$ RndKey;
  [b, _] <-$2 A (f_oracle k) tt;
  ret b.
Definition PRF_G_B : Comp bool :=
  [b, _] <-$2 A (RndR_func) nil;
  ret b.
Definition PRF_Advantage :=
  | Pr[PRF_G_A] - Pr[PRF_G_B] |.
```

Listing 26: PRF Concrete Security Definition

```
Variable eta : nat.
Variable f : Bvector eta ->
  Bvector eta -> Bvector eta.

Definition PRFE_KeyGen :=
  {0, 1} ^ eta.
Definition PRFE_Encrypt
  (k : Key )(p : Plaintext) :=
  r <-$ {0, 1} ^ eta;
  ret (r, p xor (f k r)).
Definition PRFE_Decrypt
  (k : Key)(c : Ciphertext) :=
  (snd c) xor (f k (fst c)).
```

Listing 27: Encryption using a PRF

```
Variable Plaintext Ciphertext Key State : Set.
Variable KeyGen : Comp Key.
Variable Encrypt : Key -> Ciphertext
  -> Comp Plaintext.
Variable A1 : OracleComp
  Plaintext Ciphertext
  (Plaintext * Plaintext * State).
Variable A2 : State -> Ciphertext ->
  OracleComp Plaintext Ciphertext bool.

Definition EncryptOracle
  (k: Key)(x: unit)(p: Plaintext) :=
  c <-$ Encrypt k p;
  ret (c, tt).
Definition IND_CPA_SecretKey_G :=
  key <-$ KeyGen ;
  [b, _] <-$2
  (
    [p0, p1, s_A] <--$3 A1;
    b <--$$ {0, 1};
    pb <- if b then p1 else p0;
    c <--$$ Encrypt key pb;
    b’ <--$ A2 s_A c;
    $ ret eqb b b’
  )
  (EncryptOracle key) tt;
  ret b.
Definition
  IND_CPA_SecretKey_Advantage :=
  | Pr[IND_CPA_SecretKey_G] - 1/2 |.
```

Listing 28: IND-CPA Concrete Security Definition

some game and an expression that describes the *advantage* of some adversary – *i.e.*, the probability that the adversary will “win” the game.

The game used to define the concrete security of a PRF is shown in Listing 26. Less formally, I say that f is a PRF for some adversary A if A cannot effectively distinguish f from a random function. So this means that I expect that `PRF_Advantage` is “small” as long as `A` is an admissible adversary.

The function `f_oracle` simply puts the function `f` in the form of an oracle, though a very simple one with no state and with deterministic behavior. Recall that an oracle in FCF is any term of type `S -> A -> Comp (B * S)` for arbitrary types `S`, `A`, and `B`. The procedure `RndR_func` is an oracle implementing a random function constructed using the provided computation `RndR`. The expressions involving `A` use a coercion in Coq to invoke the denotational semantics for `OracleComp`, and therefore ensure that `A` can query the oracle but has no access to the state of the oracle.

At a high level, this definition involves two games describing two different "worlds" in which the adversary may find itself. In one world (`PRF_G_A`) the adversary interacts with the PRF, and in the other (`PRF_G_B`) the adversary interacts with a random function. In each game, the adversary interacts with the oracle and then outputs a bit. The advantage of the adversary is the difference between the probability that it outputs 1 in world `PRF_G_A` and the probability that it outputs 1 in world `PRF_G_B`. If `f` is a PRF, then this advantage should be small.

The concrete security definition for IND-CPA encryption is shown in Listing 28. Note that this is the symmetric key version of this definition, so it differs from the security definition used in the El Gamal proof. In this definition, `KeyGen` and `Encrypt` are the key generation and encryption procedures. The adversary comprises two procedures, `A1` and `A2` with different signatures, and the adversary is allowed to share arbitrary state information between these two procedures. This definition uses a slightly different style than the PRF definition—there is one game and the "world" is chosen at random within that game. Then the adversary attempts to determine which world was chosen.

In Listing 28, the game produces an encryption oracle from the `Encrypt` function and a randomly-generated encryption key. Then the remainder of the game, including the calls to `A1` and `A2`, may interact with that oracle.

5.2.2 Construction

The construction, like the security definitions, can be modeled in a very natural way. Of course, one must take care to ensure that the construction has the correct signature as specified in the desired security property. The PRF encryption construction is shown in Listing 27.

In the PRF Encryption construction, I assume a `nat` called `eta` (η) which will serve as the security parameter. The encryption scheme is based on a function `f`, and the scheme will only be secure if `f` is a PRF. The type of keys and plaintexts is bit vectors of length `eta`, and the type of ciphertexts is pairs of these bit vectors. The decryption function is included for completeness, but it is not needed for this security proof.

5.2.3 SEQUENCE OF GAMES

The sequence of games represents the overall strategy for completing the proof. In the case of PRF Encryption, I want to show that the probability that the adversary will correctly guess the randomly chosen "world" is close to ½. I accomplish this by instantiating the IND-CPA security definition with the construction, and then transforming this game, little by little, until I have a game in which this probability is exactly ½. Each transformation may add some concrete value to the bounds, and I want to ensure that the sum of these values is small.

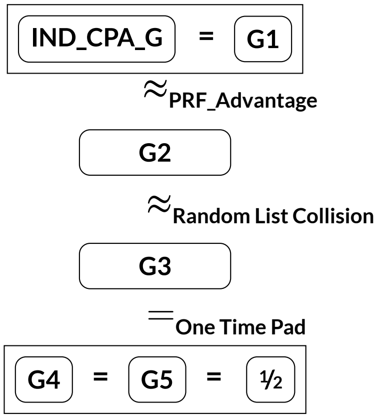


Figure 5.1: PRF Encryption Sequence of Games

```
Definition PRFE_Encrypt_OC (x : unit)
  (p : Plaintext) : OracleComp
  (Bvector eta) (Bvector eta)
  (Ciphertext * unit) :=
  r <--$$ {0,1} ^ eta;
  pad <--$ OC_Query r;
  $ (ret (r, p xor pad, tt)).

Definition PRF_A : OracleComp
  (Bvector eta) (Bvector eta) bool :=
  [a, n] <--$2 OC_Run A1 PRFE_Encrypt_OC tt;
  [p0, p1, s_A] <-3 a;
  b <--$$ {0,1}; r <--$$ {0,1}^eta;
  pb <- if b then p1 else p0;
  pad <--$ OC_Query r;
  c <- (r, pb xor pad);
  z <--$ OC_Run (A2 s_A c) PRFE_Encrypt_OC n;
  [b’,_] <-2 z; $ ret (eqb b b’).
```

Listing 29: The Constructed Adversary Against the PRF

The diagram in Figure 5.1 shows the entire sequence of games, as well as the relationship between each pair of games in the sequence. In this diagram, two games are related by $=$ if they are identical, and by $\approx$ if they are close. When the equivalence is non-trivial, the diagram gives an argument for the equivalence, which implies a bound on the distance between the games when they are not equal. The intermediate game code is omitted, but a detailed description of each game transformation follows.

I begin by instantiating the IND-CPA definition with the construction and simplifying to produce game G1. This equivalence is obvious, and the proof can be completed using Coq's `reflexivity` tactic.

Next we replace the function `f` with a random function, and the distance between G1 and G2 is exactly the advantage of some adversary against a PRF. The adversary against the PRF (Listing 29) is constructed from `A1` and `A2`. `PRFE_Encrypt_OC` is an encryption oracle that interacts with the PRF as an oracle. `PRF_A` provides this encryption oracle to `A1` and `A2` (the two adversary procedures in the IND-CPA definition) using the `OC_Run` operation. This proof can be completed by performing simple manipulations and then unifying with `PRF_Advantage`.

Now I replace the random function output used to encrypt the challenge ciphertext with a bit vector selected uniformly at random to produce game G3. I show that G2 and G3 are "close" by demonstrating that these games are "identical until bad" in the sense of Theorem 6. The "bad" event of interest is the event that the randomly-generated PRF input used to encrypt the challenge plaintext is also used to encrypt some other value during the interaction between the adversary and the encryption oracle. There are two separate adversary procedures, and each one is capable of encountering `r` during its interaction with the oracle. To get an expression for the probability of the "bad" event, I assume natural numbers q_1 and q_2, and that `A1` performs at most q_1 queries and `A2` performs at most q_2 queries. FCF includes a library module called `RndInList` that includes general-purpose arguments related to the probability of encountering a randomly selected value in a list of a certain length, and the probability of encountering a certain value in a list of randomly-generated elements of a certain length. Using these arguments, I conclude that the distance between G2 and

G3 is $q_1/2^\eta + q_2/2^\eta$.

The previous equivalences are proven using the program logic described in Chapter 4. Once the random functions are removed, there are no more issues related to state, and the remainder of the proof can be completed by reasoning on the probability distributions using the theory of distributions.

```
Theorem G3_G3_1_equiv:
  Pr[G3] == Pr[G3_1].

  unfold G3, G3_1.
  repeat (comp_simp;
    inline_first;
    comp_skip).
Qed.

Theorem G3_1_G4_equiv:
  Pr[G3_1] == Pr[G4].

    unfold G3_1, G4.
    do 4 (comp_skip;
      comp_simp).
    apply xor_OTP_eq.
    reflexivity.
Qed.
```

Listing 30: Proof of Equivalence of G3 and G4

In G3, the encryption of the challenge plaintext is by one-time pad, so I can replace the resulting ciphertext with a randomly-chosen value to produce G4 using the generic one-time pad argument provided with the FCF library. This step is relatively simple so I include the full code of the proof (Listing 30) for illustration. The one-time pad argument expects the game to be in a particular form, so I develop another intermediate game (`G3_1`), and I start by proving that `G3` is equivalent to `G3_1`. These games only differ by associativity, so a simple repeated proof script establishes their equivalence. The second proof in Listing 30 focuses on the appropriate context, and then applies the one-time pad argument for `xor`.

In G4, the challenge bit is independent of all other values in the game, so I can move the sampling of this bit to the end of the game to produce G5. The proof of equivalence is by repeated application of the commutativity theorem (Theorem 2).

Finally, I develop the proof that the adversary wins Game 5 with probability exactly ½. This proof proceeds by discarding all of the statements in the game before the coin flip. Then what remains is a very simple game that flips a coin and compares the result to a fixed value. A provided tactic can automatically determine that the probability that this game returns *true* is ½.

By combining the equivalences of each pair of intermediate games, I get the final concrete security result shown in Listing 31. It is important to note that the statement of this theorem does not reference any of the intermediate games. The sequence of games was only a tool that we used to get the final result, and this sequence does not need to be inspected in order to trust the result.

```
Theorem PRFE_IND_CPA_concrete :
  IND_CPA_SecretKey_Advantage PRFE_KeyGen PRFE_Encrypt A1 A2 <=
  PRF_Advantage ({0,1}^eta) ({0,1}^eta) f PRF_A + (q1 / 2^eta + q2 / 2^eta).
```

Listing 31: Concrete Security Result

This completes the proof of security in the concrete setting. In the next subsections, I use this result to produce a security proof in the asymptotic setting.

5.2.4 Asymptotic Security Definitions

Now I give the asymptotic security definitions for PRFs and IND-CPA encryption. These definitions are parameterized by an admissibility predicate as described in Section 4.4.1. The IND-CPA definition accepts two admissibility predicates—one for each adversary procedure.

The asymptotic security definition for a PRF is given in Listing 32. In this definition, `RndKey`, `RndR`, and `f` are `nat`-indexed families of procedures. Similarly in the IND-CPA definition (Listing 33), `KeyGen` and `Encrypt` are `nat`-indexed families of procedures. Both of these definitions are claims over all admissible `nat`-indexed adversary families. Note that both definitions reuse the expressions provided in the concrete security definitions. This style provides a convenient method of developing an asymptotic security proof from a concrete security proof.

5.2.5 Efficiency of Constructed Adversaries

The first step in proving an asymptotic security result is to view each constructed adversary in the concrete proof as a `nat`-indexed family of adversaries, and prove that this family is “efficient” as

```
Variable D R Key : nat -> Set.
Variable RndKey : forall n, Comp (Key n).
Variable RndR : forall n, Comp (R n).
Variable f : forall n, Key n -> D n-> R n.

Definition PRF :=
  forall (A : \forall n, OracleComp (D n) (R n)
    bool), admissible_A A ->
    negligible (fun n => PRF_Advantage
      (RndKey n) (RndR n) (@f n) (A n)).
```

Listing 32: Definition of a PRF

```
Variable Plaintext Ciphertext Key State :
  nat -> Set.
Variable KeyGen : forall n, Comp (Key n).
Variable Encrypt : forall n, Key n ->
  Ciphertext n -> Comp (Plaintext n).

Definition IND_CPA_SecretKey :=
  forall (State : nat -> Set)
  (A1 : forall n, OracleComp (Plaintext n)
    (Ciphertext n)
    (Plaintext n * Plaintext n * State n))
  (A2 : forall n, State n -> Ciphertext n ->
    OracleComp (Plaintext n) (Ciphertext n) bool),
    admissible_A1 A1 ->
    admissible_A2 A2 ->
    negligible
      (fun n => IND_CPA_SecretKey_Advantage
      (KeyGen n) (@Encrypt n) (A1 n) (A2 n) ).
```

Listing 33: Definition of IND-CPA Encryption

defined by some complexity class. In the PRF Encryption proof, I use the non-uniform polynomial time complexity class described in Section 4.4.1. Because this class includes a concrete cost model, I begin with a proof of the concrete cost of each constructed adversary procedure.

First I assume costs for `A1` and `A2`. `A1_cost` is a function describing the cost of `A_1`. `A2_cost_1` is a number describing how much it costs for `A2` to compute an `OracleComp` that is closed over a state and a ciphertext. Then `A2_cost_2` is a function describing the cost of executing this `OracleComp`. Given these assumptions, I can give a cost to `PRF_A` as shown in Listing 34. In the statement of this theorem, `oc_cost`, `comp_cost`, and `cost` are the cost models for `OracleComp`, `Comp`, and Coq functions, respectively. Note that this cost model is overly conservative and some costs are counted multiple times.

```
Theorem PRF_A_cost :
  oc_cost cost (comp_cost cost) PRF_A
    (fun x => (A1_cost (x + (5 * eta))) +
    (A2_cost_2 (x + (5 * eta))) +
     x + 5 * A2_cost_1 + 6 + 7 * eta).
```

Listing 34: Cost of Constructed Procedure PRF_A

This proof is completed by repeatedly applying the rule of the cost model that is relevant to the term in the goal, which is a highly syntax-directed operation that can be mostly automated. Once all these syntax-directed rules are applied, the developer is obligated to prove that the expression obtained in this process is equal to (or less than) the expression in the statement of the theorem. In this last step of the proof, automated tactics such as `omega` are very useful.

5.2.6 ASYMPTOTIC SECURITY PROOF

The final step in the proof is to show that the security definition shown in Listing 33 holds on this construction as long as `f` is a PRF as defined in Listing 32. The statement of this fact is shown in Listing 35. Note that `admissible_oc` and `admissible_oc_func_2` are the admissibility predicates for `OracleComp` and for functions with two arguments that produce an `OracleComp` defined in the complexity class.

```
Theorem PRFE_IND_CPA :
  PRF Rnd Rnd f (admissible_oc cost) ->
  IND_CPA_SecretKey
    PRFE_KeyGen (fun n => PRFE_Encrypt (@f n))
    (admissible_oc cost)
    (admissible_oc_func_2 cost).
```

Listing 35: Asymptotic Security of PRF Encryption

The primary obligation of this proof is to show that the function defining the advantage of any admissible family of adversaries against this encryption scheme is a negligible function. The fact that this adversary family is admissible allows us to use the result of Listing 34, along with other facts, to conclude that the constructed adversary family against the PRF is admissible. In the course of this proof, I must show that the expression implied by Figure 34 is at most polynomial in η if x is at most polynomial in η and all the costs related to `PRF_A1` and `PRF_A2` are at most polynomial in η. This fact is proven using the provided theory of polynomial functions (Section 4.4).

From the admissibility of the constructed adversary, and from the fact the `f` is a PRF against all admissible adversaries, I can conclude that the constructed adversary's advantage against the PRF is negligible. The advantage of this adversary against the PRF is one of the terms that appears in the bounds of the concrete result (Listing 31). The other term is $q_1/2^\eta + q_2/2^\eta$, where q_1 and q_2 are the number of oracle queries performed by the two adversary procedures. The admissibility predicates ensure that each adversary only performs a polynomial number of queries, so q_1 and q_2 must be polynomial in η, and this expression is negligible in η. So the advantage of the adversary against this encryption scheme is the sum of two negligible functions, and is therefore negligible.

5.2.7 Proof Engineering

The entire proof of security for this encryption scheme requires approximately 1500 lines of Coq code, of which about 700 lines are specification (including 100 lines of cryptographic definitions and intermediate games) and 800 lines are proof. The proof incorporates another 500 lines of code for the reusable arguments (*e.g.*, the one-time pad argument). I expect that a skilled Coq developer could complete such a proof in a matter of days (though he may require the help of a cryptographer to develop the sequence of games and high-level arguments). Though this proof is relatively simple, it includes several elements that one would find in a typical cryptographic proof, and it is a good basis for estimating the effort required to complete a more complex proof.

5.3 A NEGATIVE EXAMPLE: DUAL_EC_DRBG

In this section, I mechanize the proof of Brown and Gjøsteen[22] that Dual EC DRBG is a cryptographic pseudorandom generator (PRG). This PRG was standardized in ANSI X9.82 and NIST SP 800-90A in 2005 and 2006, respectively. In 2007, Shumow and Ferguson described[43] how Dual EC DRBG possibly contains a "back door" that would give certain parties the ability to easily predict the output of the PRG, thus defeating its security.

It is not uncommon for a single scheme to be proven secure and known to be vulnerable at the same time, and this conflict is typically caused by a mismatch between the model used in the proof and the realization of the construction or the adversary. In the case of Dual EC DRBG, the proof of Brown and Gjøsteen uses a slightly idealized form of the construction, which is not the same as the construction published in the ANSI and NIST standards. I will present the proof of security of the idealized form of this scheme, then modify the construction in order to match the standardized version. I will then show that the proof of security is no longer valid, and I will argue that no proof of security exists for the standardized version of this scheme. This exercise illustrates the importance of inspecting the models used in the proof, and it shows how FCF can be used to locate vulnerabilities in insecure schemes.

5.3.1 DUAL EC DRBG SECURITY

Informally, a PRG is a scheme that produces a number of pseudorandom bits from a fixed random seed. The PRG has some state, and it provides a function which produces some output and a new value for its state. By calling this function repeatedly, it should be possible to produce an arbitrary (polynomial) number of pseudorandom bits.

Dual EC DRBG is based on a finite cyclic group, and both the generator state and the output is an element of this group. In reality, this group is derived from an elliptic curve over a finite field, but

I can complete this proof of security using the finite cyclic group type class shown in Section 5.1. I also assume the functions `x` and `from_x` which converts a group element to a natural number and produces a group element from a natural number, respectively. Because the group is based on an elliptic curve, these functions model the operation of converting to/from a group element using the value of the x coordinate.

The scheme relies on two group element parameters `P` and `Q`. The construction is shown in Listing 36. The function in this listing also takes an additional `nat` parameter that defines the random seed. The security definition is provided in Listing 37. Because this is a simple exercise, I use a security definition that is specialized to this scheme, and this definition matches the security definition provided by Brown and Gjøsteen. In this definition, an adversary that has not knowledge of the initial state of the PRG should be unable to distinguish the new PRG state and output from uniformly random group elements.

In Listing 37, `P` is a fixed global parameter, and `Q` is selected at random. The fact that `Q` is random is of critical importance to this proof. In the standardized version of this scheme, `Q` is a fixed global parameter instead of a randomly-selected value. I designed the model so that `Q` is a parameter to the construction and security definition, and therefore I can use the same functions in both versions of this model. The function `DRBG_P` provides the idealized version of the model by generating `Q` at random.

```
Definition DRBG(P Q : GroupElement)(t : nat) : (GroupElement * GroupElement ) :=
  let s :=  x (P ^ t) in (P ^ s, Q ^ t).
```

Listing 36: Dual EC DRBG Construction

The security of Dual EC DRBG is based on the hardness of the decisional Diffie-Hellman (DDH) problem and a variant of the discrete logarithm problem (DLP). In order to focus on the relevant parts of this exercise, I define an intermediate game `G2` and simply declare that the distance between this game and `DRBG_GA` is equal to the advantage of some adversary against this variant of the DLP.

```
Definition DRBG_GA Q :=
  seed <-$ RndNat order;
  [s1, v1] <-2 DRBG P Q seed;
  b <-$ A (Q, s1, v1);
  ret b.

Definition DRBG_GB Q :=
  x2 <-$ RndNat order;
  x3 <-$ RndNat order;
  b <-$ A (Q, (P^x2), (P^x3));
  ret b.

Definition DRBG_P (f : GroupElement -> Comp bool) :=
  x <-$ RndNat order;
  Q <- P ^ x;
  f Q.

Definition DRBG_Advantage := | Pr[DRBG_P DRBG_GA] - Pr[DRBG_P DRBG_GB] |.
```

Listing 37: Security Definition for Dual EC DRBG

This intermediate game is shown in Listing 38.

```
Definition G2 Q :=
  seed <-$ RndNat order;
  [s1, v1] <-2 (seed, Q ^ seed);
  b <-$ A (Q, (P ^ s1), v1);
  ret b.

Definition xLogAdvantage := | Pr[DRBG_P DRBG_GA] - Pr[DRBG_P G2] |.
```

Listing 38: Dual EC DRBG Intermediate Game and DLP Definition

Now I can show that the distance between this intermediate game and `DRBH_GB` is equal to the advantage of some adversary against the DDH problem. The statement of this theorem and the final security result for this scheme are shown in Listing 39.

```
Theorem DRBG_P_DH : | Pr[DRBG_P G2] - Pr[DRBG_P DRBG_GB] | ==
  DDH_Advantage _ groupOp ident inverse _ g order A.

Theorem DRBG_P_secure :  | Pr[DRBG_P DRBG_GA] - Pr[DRBG_P DRBG_GB] | <=
  xLogAdvantage + DDH_Advantage _ groupOp ident inverse _ g order A.
```

Listing 39: Dual EC DRBG Security

This completes the proof. Now I turn my attention to the standardized version of this scheme, in which `Q` is a global parameter rather than being chosen at random. To model this variant, I simply use the function `DRBG_S` that specializes some other definition using this fixed value of `Q`. Then I try to prove that the distance between `G2` and `DRBG_GB` is still equal to the DDH advantage. These items are shown in Listing 40.

```
Definition DRBG_S (f : GroupElement -> Comp bool) := f Q.

Theorem DRBG_S_DH : | Pr[DRBG_S G2] - Pr[DRBG_S DRBG_GB] | ==
  DDH_Advantage _ groupOp ident inverse _ g order A.
```

Listing 40: Standardized Variant of Dual EC DRBG

Of course, the proof from the idealized scheme simply does not work here. In order to unify with the DDH definition, `Q` must be generated at random. As a result of this mismatch, there can be no proof of the statement shown in Listing 40. This means there is no way to reduce the security of this scheme to the DDH, but there may still be some other reduction that is still possible.

If a person was trying to complete this proof, the failure to prove the theorem in Listing 40 should be illuminating. The inability to prove this fact may actually stem from a weakness in the scheme. The developer may then wonder if the scheme really is secure for all choices of `P` and `Q`. This is an incredibly strong statement, and the developer would probably suspect that there is some choice of these parameters that renders this scheme insecure. In fact, Shumow and Ferguson describe a way in which the parameters can be carefully chosen that gives the party that chooses the parameters the ability to determine the state of the PRG and determine its output.

5.4 Conclusion

In this chapter, I provided several complete examples that illustrate how FCF can be used to develop proofs of security for cryptographic schemes, and an example that demonstrates how FCF can be used to locate flaws in such schemes. These are all relatively simple examples, and Chapter 6 contains a complete proof for a complex searchable symmetric encryption scheme.

6

Searchable Symmetric Encryption

This chapter demonstrates the viability of using FCF to construct formal proofs of security for complex cryptographic schemes by proving the security of the efficient Searchable Symmetric Encryption (SSE) scheme of Cash et al.[25] Using this SSE scheme, a client can store a large database on an untrusted server, and the server can efficiently query the database on behalf of the client without learning the contents of the database or the query. This scheme is accompanied by a proof of security on paper, but we can gain greater assurance of the security of this scheme by developing a mechanized proof of security in FCF. Note that the scheme we verified in this effort is exactly the scheme described by Cash et al., and my formal proof was inspired by the proof presented in the paper.

Following the release of EasyCrypt[9], a team of cryptographers and programming language experts attempted[35] to prove the security of a Private Information Retrieval (PIR) system[29] which can be viewed as a predecessor to the SSE scheme of Cash et al. This effort did not produce a complete proof because certain required facts could not be proven in EasyCrypt. Specifically, it was impossible at the time to prove particular equivalences involving loop fusion and order permutation within a loop without modifying the EasyCrypt code to accept these equivalences.

EasyCrypt has seen significant improvement since its release, and a proof of security for a greatly simplified form of this PIR scheme[44] has been completed in EasyCrypt. In parallel, FCF was developed in order to find a more general solution to the problem of "missing" theory in cryptographic frameworks such as EasyCrypt. Due to the foundational nature of FCF, any required theorem can be formally derived from the semantics without increasing the trusted computing base. I rely on this trustworthy extensibility of FCF to develop the additional theory required to complete the proof described in this paper.

The proof described here is among the most complex mechanized cryptographic proofs that have been developed to date. Table 6.1 (in Section 6.4) summarizes the complexity of this proof, which comprises several cryptographic reductions including over 14,000 lines of Coq code and 58 intermediate games. This development effort also produced a significant amount of FCF theory related to loop transformations, hybrid arguments, sampling without replacement, and constructions involving repeated independent trials. I added this theory to the standard library of FCF in order to assist with future proof development efforts.

6.1 SEARCHABLE SYMMETRIC ENCRYPTION PROOF OVERVIEW

This section informally introduces Searchable Symmetric Encryption and describes the strategy used in the proof of security. An SSE scheme provides a mechanism to encrypt a database and a list of queries. These encryptions are given to an untrusted party who is able to produce encryptions of the result of executing the queries on the database while learning very little about the database or queries. We call the party that knows the unencrypted database the *client*, and the untrusted party that carries out queries on behalf of the client is the *server*. A database is simply a list of identifiers and a set of keywords associated with each identifier. Each identifier can be used to retrieve some other object in an encrypted database, but this operation is beyond the scope of the SSE definitions.

The SSE scheme is constructed from an abstraction called a Tuple Set (or T-Set) that behaves like a secure associative array. In this proof, I consider single-keyword SSE (SKS-SSE), in which each query is a single keyword. Roughly speaking, this scheme works by encrypting each value using a key derived from the appropriate keyword, and then storing the ciphertexts in a T-Set. The server can perform a query by looking up the specified keyword in the T-Set and giving the resulting ciphertexts to the client. Cash et al. describe several variants of their SSE scheme which support increasingly sophisticated queries, and SKS-SSE is the simplest of these variants.

Figure 6.1 describes the structure of the security proof. Each node in the diagram is an object that is conjectured (in the case of PRF) or proved to exist, and each arrow is a reduction that proves the existence of some construction. Many of these reductions are complex arguments involving large sequences of games. In particular, the T-Set construction and the proofs related to this construction are quite complex, and the T-Set abstraction hides the complexity of this construction in order to make the SSE proof simpler. This is a standard technique in cryptography that is even more important when developing mechanized proofs. The abstraction and modular construction features of Coq, which are inherited by FCF, are very useful for developing these sorts of proofs.

The left side of the diagram shows the proof that the T-Set construction is secure and correct, and the right side is the proof of security of the SKS-SSE scheme. In the T-Set proof, I begin by showing that if some function f is a PRF, then it is an *iterated* PRF as described in Section 3.3. From a PRF and an iterated PRF, I show that a simplified "single-trial" form of the T-Set construction is correct and secure. Then I use some reusable arguments to obtain the correctness and security of the "full" T-Set construction. More information about the T-Set security and correctness proofs can be found in Sections 6.3.1 and 6.3.2, respectively.

The proof of security for SKS-SSE requires an IND-CPA encryption scheme, which can be formally derived from a PRF as shown in Section 5.2. I then show that this encryption scheme is an *iterated* encryption scheme in a manner similar to the iterated PRF reduction. This fact also follows from the hybrid argument described in Section 3.3. The I show that the SKS-SSE scheme is secure as long as the T-Set is correct and secure, the encryption scheme used is an iterated IND-CPA encryption scheme, and the function used to derive encryption keys is a PRF. I expand on this part of the proof in Section 6.2.

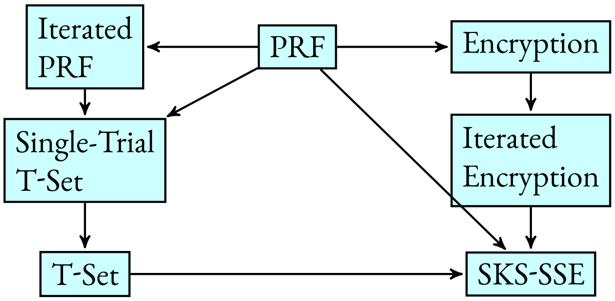


Figure 6.1: SSE Security Proof Structure

6.2 SINGLE KEYWORD SEARCHABLE SYMMETRIC ENCRYPTION FROM TUPLE SETS

In this section, I present the formal definitions related to SSE and Tuple Sets, and formally prove the security of the SKS-SSE scheme of Cash et al. An SSE scheme consists of an `EDBSetup` function that takes a database and produces an encrypted database and a key, and a `SearchProtocol` that uses a key and a query known to the client and an encrypted database known to the server to produce a list of identifiers and a transcript.

6.2.1 Non-Adaptively Secure SSE

I use a non-adaptively secure definition for SSE (Listing 41), in which an adversary produces a database and the entire list of queries up front. The definition is given as an indistinguishability between a pair of games parameterized by a leakage function `L`. The leakage function describes what information is allowed to leak to the adversary, and this function must be inspected carefully in order to determine if the leakage is acceptable. The real game uses the actual `EDBSetup` and `SearchProtocol` while the ideal game uses a simulator that is only given the result of the leakage function applied to the unencrypted database and list of queries. The SSE scheme is non-adaptively secure if the distance between these two games, `SSE_NA_Advantage`, is small.

In this definition, the adversary is divided into two separate procedures, `A1`, and `A2` which are allowed to share state. In the corresponding definition provided by Cash et al., the second adversary procedure is also given the list of identifiers resulting from the queries in order to model the assumption that the client will immediately give the identifiers to the server to retrieve the required objects. For simplicity, I remove this assumption and only give the search transcript to the adversary. Because the correct identifiers are already known to the adversary, these definitions are equivalent under the assumption that the SSE scheme is (computationally) correct.

6.2.2 T-Sets

A T-Set is a primitive that associates values with keywords, and allows retrieval of all the values associated with some keyword. A T-Set differs from a standard associative array in that the T-Set scheme

```
Definition SSE_Sec_NA_Real :=
  [db, q, s_A] <-$3 A1;
  [k, edb] <-$2 EDBSetup db;
  ls <-$ foreach (x in q) (SearchProtocol edb k x);
  A2 s_A edb (snd (split ls)).

Definition SSE_Sec_NA_Ideal :=
  [db, q, s_A] <-$3 A1;
  leak <-$ L db q;
  [edb, t] <-$2 Sim leak;
  A2 s_A edb t.

Definition SSE_NA_Advantage :=
  | Pr[SSE_Sec_NA_Real] - Pr[SSE_Sec_NA_Ideal] |.
```

Listing 41: SSE Non-Adaptive Security

attempts to hide as much as possible about the values in the T-Set and the relationship between keywords and values. A server that possesses a T-Set structure but not the key for that structure should learn very little about the contents of the structure. The server can also query the structure on behalf of a client that knows the T-Set key, and in the process the server should learn very little other than the set of values returned by the query.

A T-Set scheme is composed of three procedures: `TSetSetup`, `TSetGetTag`, and `TSetRetrieve`. `TSetSetup` takes a database and returns a T-Set and a secret key. Database keywords are elements of $\{0, 1\}^*$ and identifiers are elements of $\{0, 1\}^\lambda$. `TSetGetTag` takes a secret key and outputs a tag. `TSetRetrieve` takes a T-Set and a tag and returns a list of identifiers.

The security of the SSE scheme relies on both the security and the correctness of the T-Set scheme. The formal correctness definition (Listing 42) is computational and non-adaptive. In this definition, the adversary chooses the database and list of keywords, and the correct answers are compared to the answers produced using the T-Set. If the T-Set is correct, then the probability that these answers differ (`AdvCor`) is small.

The non-adaptive security of a T-Set scheme is defined as a real/ideal indistinguishability parameterized by a leakage function `L` as shown in Listing 43. If the T-Set is secure, then `TSetAdvantage`

```
Definition AdvCor_G :=
  [t, q] <-$2 A;
  [tSet, k_T] <-$2 TSetSetup t;
  tags <-$ foreach (x in q) (TSetGetTag k_T x);
  t_w <- foreach (x in tags) (TSetRetrieve tSet x);
  t_w_correct <- foreach (x in q)
    (arrayLookupList _ t x);
  ret (t_w != t_w_correct).

Definition AdvCor := Pr[AdvCor_G].
```

Listing 42: T-Set Non-Adaptive Computational Correctness

should be small. Note that the correct answers are given to the simulator in the ideal game, implying that this information is allowed to leak to the adversary. The T-Set only hides information about the queries and the *non-queried* portions of the database.

```
Definition TSetReal :=
  [t, q, s_A] <-$3 A1;
  [tSet, k_T] <-$2 TSetSetup t;
  tags <-$ foreach (x in q) (TSetGetTag k_T x);
  A2 s_A (tSet, tags).

Definition TSetIdeal :=
  [t, q, s_A] <-$3 A1;
  T_qs <- foreach (x in q) (lookupAnswers t x);
  [tSet, tags] <-$2 Sim (L t q) T_qs;
  A2 s_A (tSet, tags).

Definition TSetAdvantage :=
  | Pr[TSetReal] - Pr[TSetIdeal] |.
```

Listing 43: T-Set Security Definition

6.2.3 IND-CPA Encryption and PRFs

The final elements required to construct the SSE scheme are an IND-CPA encryption scheme and a pseudorandom function. The T-Set is allowed to leak information about values returned by queries, so the SSE scheme stores ciphertexts in the T-Set instead of indices. Because the encryption is IND-CPA, the only information leaked is the number of values returned by each query. The encryption key is determined by a pseudorandom function applied to the appropriate keyword. I use adaptively-secure encryption and PRFs in this proof merely for convenience, and it would be possible to complete this proof using non-adaptive forms of these assumptions.

The particular IND-CPA definition that is used as an assumption is shown in Listing 44. In this definition, `EncryptOracle` is an oracle that returns an encryption of any plaintext it receives, and `EncryptNothingOracle` takes a plaintext and returns an encryption of some default value (e.g. 0^λ). The scheme encrypts each entry using a key derived from the keyword, so the proof actually requires an iterated form of IND-CPA in which the adversary is allowed to interact with several encryption oracles, each with a different key. I can show that any IND-CPA encryption scheme is also an iterated IND-CPA encryption scheme (security definition omitted) using the hybrid argument described in Section 3.3. The adaptively-secure PRF definition used in the proof is shown in Listing 45.

```
Definition IND_CPA_SK_O_G0 :=
  key <-$ KeyGen;
  [b, _] <-$2 A (EncryptOracle key) tt;
  ret b.

Definition IND_CPA_SK_O_G1 :=
  key <-$ KeyGen;
  [b, _] <-$2 A (EncryptNothingOracle key) tt;
  ret b.

Definition IND_CPA_SK_O_Advantage :=
  | Pr[IND_CPA_SK_O_G0] - Pr[IND_CPA_SK_O_G1]  |.
```

Listing 44: Iterated IND-CPA Encryption

6.2.4 SKS-SSE Construction

The formalization of the SKS-SSE construction is shown in Figure 46. In this figure, `Enc` and `Dec` are the encryption and decryption procedures for an IND-CPA encryption scheme, and `F` is a PRF. The `EDBSetup` routine iterates over all keywords in the database (obtained using the `toW` function) and encrypts the indices associated with each keyword under a key derived from that keyword.

```
Definition f_oracle(k : Key)(x : unit)(d : D) :=
  ret (f k d, tt).

Definition PRF_G_A : Comp bool :=
  k <-$ RndKey;
  [b, _] <-$2 A (f_oracle k) tt;
  ret b.

Definition PRF_G_B : Comp bool :=
  [b, _] <-$2 A (RndR_func) nil;
  ret b.

Definition PRF_Advantage :=
  | Pr[PRF_G_A] - Pr[PRF_G_B] |.
```

Listing 45: Adaptively-Secure PRF

Then `TSetSetup` is used to construct a T-Set from this encrypted database. In this procedure, `lookupInds` returns all the indices associated with a keyword. The search protocol uses `TSetGetTag` and `TSetRetrieve` to get the encrypted indices, and then decrypts them.

```
Definition SKS_EDBSetup_wLoop db k_S w :=
  k_e <- F k_S w;
  inds <- lookupInds db w;
  t <-$ foreach (x in inds) (Enc k_e x);
  ret (w, t).

Definition SKS_EDBSetup(db : DB) :=
  k_S <-$ {0, 1}^lambda;
  t <-$ foreach (x in (toW db))
    (SKS_EDBSetup_wLoop db k_S x);
  [tSet, k_T] <-$2 TSetSetup t;
  ret ((k_S, k_T), tSet).

Definition SKS_Search tSet k w :=
  [k_S, k_T] <-2 k;
  (* client *) tag <-$ TSetGetTag k_T w;
  (* server *) t <- TSetRetrieve tSet tag;
  (* client *) inds <- map (Dec (F k_S w)) t;
  ret (inds, (tag, t)).
```

Listing 46: SKS-SSE Construction

6.2.5 PROOF OF SECURITY FOR SKS-SSE

Listing 47 contains the leakage function and simulator used in the proof of security. Note that `L_T` is the leakage function for the T-Set. Informally, this scheme leaks the number of indices associated with each queried keyword, as well as the result of the T-Set leakage function applied to the *structure* of the database (which is essentially the number of indices associated with each keyword) and the list of queries. The simulator for this proof uses `Sim_T`, which is the T-Set simulator. In this listing, `zeroVector lambda` is a vector of length `lambda` containing all zeroes, and `combine` is the Coq function that converts a pair of lists to the corresponding list of pairs.

```
Definition SKS_resultsStruct db w :=
  k_e <-$ {0, 1}^lambda;
  inds <- lookupInds db w;
  foreach (_ in inds)
    (Enc k_e (zeroVector lambda)).

Definition L (db : DB) (qs : list Query) :=
  t_s<-$ foreach (x in (toW db))
    (SKS_resultsStruct db x);
  t <- combine (toW db) t_s;
  leak_T <- L_T t qs;
  ret (leak_T, map (arrayLookupList t) qs).

Definition SKS_Sim leak :=
  [leak_T, struct] <-2 leak;
  [tSet, tags] <-$2 Sim_T leak_T struct;
  ret (tSet, (combine tags struct)).
```

Listing 47: Leakage Function and Simulator for SKS-SSE Proof

The security proof is completed using a sequence of games (omitted). The exact security result is provided in Listing 48. The result refers to procedures `TSetCor_A`, `TSetSec_A1`, `TSetSec_A2`, `PRF_A`, `Enc_A1`, and `Enc_A2` (all omitted), which form the constructed adversaries against T-Set correctness and security, the PRF, and the IND-CPA encryption scheme. `Enc_A1` is a family of procedures, and the hypothesis states that `IND_CPA_Adv` is an upper bound on the advantage of all procedures in this family. The term `maxKeywords` represents the maximum number of keywords that may be contained in the database and queries produced by `A1`, and this term appears in the bounds due to the application of the hybrid argument as described in Section 6.2.3.

```
Theorem SKS_Secure :
  (forall i, IND_CPA_SK_O_Adv ({0, 1}^lambda) Enc
    (Enc_A1 i) Enc_A2 <= IND_CPA_Adv) ->
  SSE_NA_Advantage SKS_EDBSetup
    SKS_Search A1 A2 L SKS_Sim <=
  AdvCor TSetSetup TSetGetTag TSetRetrieve
    TSetCor_A +
  TSetAdvantage TSetSetup TSetGetTag L_T
    TSetSec_A1 TSetSec_A2 Sim_T +
  PRF_Advantage (Rnd lambda) (Rnd lambda) F PRF_A +
  maxKeywords * IND_CPA_Adv.
```

Listing 48: Exact Security of SKS-SSE Scheme

6.3 Tuple Set Instantiation

This section describes the the efficient T-Set instantiation provided by Cash et al. as well as the formal proof of security and correctness of this construction. I slightly simplify the model of the T-Set construction because I only prove non-adaptive security of the scheme. Instead of two PRFs and a random oracle, I model the scheme using only two PRFs. The random oracle is included to provide adaptive security, and it is only used when composed with one of the other functions that I model as a PRF. I can simplify the model by combining these two functions into one and assuming that the function is a PRF.

The T-Set is a hash table with B buckets, each with at most S entries. The parameters B and S are selected based on the size of the input structure T in a way that the probability of constructing the T-Set without running out of space in any bucket is non-negligible. A PRF F is used to determine the bucket into which each value will be placed, as well as a label that can be used to determine the keyword associated with the value, and a key used to encrypt the value when it is placed in the T-Set. Another PRF $\bar{F}$ is used to map keywords to tags. The security of the T-Set scheme is derived from the assumed indistinguishability of F and $\bar{F}$ from random functions.

```
Definition TSetSetup_tLoop stag length acc e :=
  [tSet, free] <- acc;
  [i, s_i] <- e; [b, L, K] <- F stag i;
  free_b <- nth b free nil;
  j <-? ($ free_b) ;
  free <- replace free b (remove free_b j);
  bet <- (S i) != length;
  newRecord <- (L, (bet :: s_i) xor K);
  tSet <- tSetUpdate tSet b j newRecord;
  ret (tSet, free).

Definition TSetSetup_wLoop T k_T acc w :=
  [tSet, free] <- acc;
  stag <- F_bar k_T w;
  t <- lookupAnswers T w;
  ls <- combine (allNatsLt (length t)) t;
  loop_over ((tSet, free), ls)
    (TSetSetup_tLoop stag (length t)).

Definition TSetSetup_trial T  :=
  k_T <-$ {0, 1} ^ lambda;
  loopRes <-$ loop_over ((nil, initFree), (toW T))
    (TSetSetup_wLoop T k_T) ;
  ret (loopRes, k_T).

Definition TSetSetup t :=
  [res, k_T] <-$ Repeat (TSetSetup_trial t)
    (fun p => isSome (fst p));
  ret (getTSet res, k_T).
```

Listing 49: T-Set Setup Routine

In order to organize the presentation and proof, I separate the TSetSetup routine into a number of subroutines. This routine is composed of a nested loop, so I provide a procedure for each loop body. Each loop body is a function that takes an accumulator and the next input value and returns the resulting value for the accumulator. The `loop_over` operator is simply notation for folding the procedure over some input list. The setup routine may fail if some bucket in the hash table is filled, so the setup is repeated in independent trials until a trial succeeds. In this listing, `nth` is a Coq function that returns the value at a certain position in a list, `remove` removes the first occurrence of some value in a list, `replace` replaces the value in a list at a specified position with another value, `tSetUpdate` sets the value in the T-Set at the specified location to the provided value, `lookupAnswers` returns the indices associated with some keyword in the T-Set, `allNatsLt` returns all the natural numbers (in increasing order) less than a specified number, and `initFree` initializes a "free list" that is used to keep track of which locations in each bucket are unoccupied. The `($ free_b)` expression in the `TSetSetup_tLoop` construction denotes sampling from the distribution corresponding to the list `free_b`. This sampling routine and notation are provided by the FCF standard library. Because this sampling may fail if the list is empty, the function perform the sampling inside a *Maybe* monad as indicated by the arrow `<-?`, and the `TSetSetup_tLoop` returns a value in an `option` type.

The `TSetGetTag` procedure (Listing 50) simply produces a tag for a keyword using the $\bar{F}$ PRF and the key for the T-Set.

```
Definition TSetGetTag (k_T : Bvector lambda) w :=
    ret (F_bar k_T w).
```

Listing 50: T-Set Get Tag Routine

```
Definition TSetRetrieve_tLoop tSet stag i :=
  [b, L, K] <-3 F stag i;
  B <- nth b tSet nil;
  t <- arrayLookupOpt _ B L;
  match t with
    | None => None
    | Some u =>
        v <- u xor K;
        bet <- Vector.hd v;
        s <- Vector.tl v;
        Some (s, bet)
  end.

Fixpoint TSetRetrieve_h tSet stag i (fuel : nat) :=
  match fuel with
    | O => nil
    | S fuel' =>
        match (TSetRetrieve_tLoop tSet stag i) with
          | Some (v, bet) =>
              v :: (if (bet) then (TSetRetrieve_h tSet stag (S i) fuel') else nil)
          | None => nil
        end
    end.

Definition TSetRetrieve tSet stag :=
  TSetRetrieve_h tSet stag O maxMatches.
```

Listing 51: T-Set Retrieve Routine

The `TSetRetrieve` procedure (Figure 51) searches through the T-Set to find all the entries matching a keyword. Because Coq requires me to model this procedure as a total function, I assume that there is a maximum number of entries (`maxMatches`) for any keyword, and we use this number as “fuel”. The loop body searches for the i_{th} value matching the tag, and returns an optional value and an indication of whether there are additional entries matching the tag. This loop body is iterated until it indicates that there are no more values, or it runs out of fuel.

6.3.1 T-Set Security

The simulator used in the security proof is shown in Listing 52. This proof is complicated by the fact that the real setup routine and the simulator perform multiple trials in an attempt to create the T-Set. So I begin by proving the security of a modified form of the scheme in which only one attempt is made to construct the T-Set. Then I combine this result with some additional arguments in order to obtain the proof of security for the full T-Set scheme.

Single-Trial T-Set Security

The Single-Trial T-Set security proof is a straightforward, though complicated, sequence of games in which I replace PRFs with random values and use the resulting randomness to show that the output is independent from the input. The first complication relates to applying the PRF definition to F in that some of the PRF keys are the same as the tags that are given to the adversary at the end of the computation. The PRF definition only applies when the PRF key is not given to the adversary, so I must split the T-Set initialization procedure into two parts: first it adds entries related to the keywords that are queried by the adversary, then it adds the rest of the entries. The first part of this procedure already matches the ideal functionality, and I only apply the PRF assumption to the entries created during the second part of the procedure. Another complication is that the initializa-

```
Definition randomTSetEntry acc :=
  label <-$ {0, 1} ^ lambda;
  value <-$ {0, 1} ^ (S lambda);
  [tSet, free] <- acc;
  b <-$ [0 .. B);
  free_b <- nth b free nil;
  j <-? ($ free_b);
  free <- replace free b (remove free_b j) nil;
  tSet <- tSetUpdate tSet b j (label, value);
  ret (tSet, free).

Definition TSetSetup_Sim_wLoop tSet_free e :=
  [tSet, free] <- tSet_free;
  [stag, t] <- e;
  ls <- combine (allNatsLt (length t)) t;
  loop_over ((tSet, free), ls)
    (TSetSetup_tLoop stag (length t)).

Definition TSet_Sim_trial n ts :=
  tags <-$ foreach (_ in ts) ({0, 1} ^ lambda);
  loopRes <-$ loop_over
    ((nil, initFree), (combine tags ts))
    TSetSetup_Sim_wLoop;
  loopRes <-$ loop_over
    (loopRes, allNatsLt (n - length (flatten ts)))
    (fun acc i => randomTSetEntry acc);
  ret (loopRes, tags).

Definition TSet_Sim leak ls :=
  [_, ts] <- split ls;
  [trialRes, tags] <-$
    Repeat (TSet_Sim_trial leak ts)
    (fun p => isSome (fst p));
  ret (getTSet trialRes, tags).
```

Listing 52: T-Set Simulator

tion procedure places each record in a random location in the correct bucket. So it is necessary to perform game manipulations in the presence of sampling *without replacement*, and the games must keep track of the unused locations in each bucket.

The intermediate game code is omitted, but a diagram of the sequence is provided in Figure 6.2. The box around the top half marks a portion of the proof that is reused as an argument in the correctness proof described in Section 6.3.2. Each equivalence in the diagram is labeled to indicate the argument or assumption used. Equivalences labeled S are simple transformations such as unfolding definitions, inlining statements, and removing unused values or statements. F indicates a loop

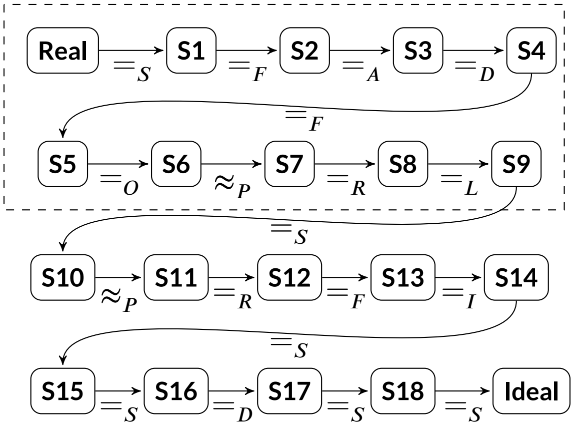


Figure 6.2: Single-Trial T-Set Security Games

fission transformation such as the one described in Chapter 3. A describes an information augmentation transformation in which additional information is added to a data structure without changing the results of the game. Such a transformation enables "ghost state" reasoning in which this additional information can be used in program logic judgments. For example, a list of ciphertexts could be augmented with a list of plaintexts and keys used in the encryption. Then a program logic judgment could state that the plaintext is equal to the value obtained by decrypting the ciphertext with the key. D is a dimension reduction where a data structure of dimension n is represented using a data structure of dimension $n - 1$. A dimension reduction may be performed to replace a 2-dimensional data structure with a list in order to apply a theorem related to list processing. O is a non-trivial change to the order in which statements are executed in the game. The T-Set construction stores entries in a random location in each bucket, requiring sampling without replacement to determine the location of each entry. In some transformations, I change the order that entries are added to the T-Set in the presence of this sampling without replacement. R equivalences replace random function outputs with independent random values by showing that there are no duplicates in the input to the function. In L transformations, I show that folding the function f over a list is equivalent to folding f over the first n elements of the list, and then folding f over the rest of the list. *I* equivalences show that certain values are independent of each other by applying a one-time pad argument.

The statement of security for single-trial T-Sets is shown in Listing 53. In this listing, `TSetSetup_once` and `TSet_Sim_once` are procedures that try to create a T-Set in a single attempt using the corresponding trial routines. These routines produce an empty T-Set if the trail fails. The procedures `TSet_PRF_A1`, `TSet_PRF_A2`, `TSet_IPRF_A1`, and `TSet_IPRF_A2`, are efficient adversaries against the PRFs constructed from `A1` and `A2`. The proof uses an iterated PRF as described in Chapter 3, and `TSet_IPRF_A1` and `TSet_IPRF_A2` form a family of adversaries constructed using different distributions from the appropriate hybrid distribution family. This theorem assumes that `F_Adv` is an upper bound on the advantage of all of these adversaries against the PRF `F`. The theorem also assumes that `F_bar_Adv` is an upper bound on the advantage of a particular constructed adversary against the PRF `F_bar`. Similar to the proof in Section 6.2.5, the database and queries provided by the adversary contain at most `maxKeywords` keywords, and this term appears in the bounds due to the application of the hybrid argument.

```
Theorem TSet_once_secure :
  (forall i, PRF_NA_Advantage
    ({0,1}^lambda) (RndF_Range) F
    (TSet_IPRF_A1 i) TSet_IPRF_A2 <= F_Adv) ->
  PRF_NA_Advantage
    ({0,1}^lambda) ({0,1}^lambda) F_bar
    TSet_PRF_A1 TSet_PRF_A2 <= F_bar_Adv ->
  TSetAdvantage TSetSetup_once TSetGetTag
    L_T TSet_Sim_once A1 A2 <=
  <= F_bar_Adv + maxKeywords * F_Adv.
```

Listing 53: Single-Trial T-Set Security

THE "ONE TO MANY" ARGUMENT

I employ a couple of non-trivial reusable arguments in order to derive security of the full T-Set scheme from the proof of security of the *Single Trial T-Set* scheme. The first of these arguments is the "One to Many" argument (Listing 54), which is a special case of the hybrid argument described in Section 3.3 in which the same argument is repeated a fixed number of times and the results are collected in a list.

```
Definition DistMult_G(c : A -> Comp B) :=
  [a, s_A] <-$2 A1;
  b <-$ foreach (x in (forNats n)) ((c a);
  A2 s_A b.

Definition DistMult_Adv :=
  | Pr[DistMult_G c1] - Pr[DistMult_G c2] |.

Theorem DistSingle_impl_Mult :
  DistMult_Adv c1 c2 A1 A2 n <=
    n * (DistSingle_Adv c1 c2 B1 B2).
```

Listing 54: The One to Many Theoreom

The "Many to Core" Argument

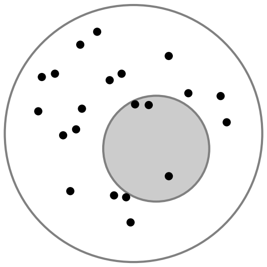


Figure 6.3: Illustration of "Many to Core" Argument

The next argument applies to any pair of probabilistic computations c_1 and c_2 that produce values of type $\boldsymbol{B}$. There is also some predicate $\boldsymbol{P}$ on values of type $\boldsymbol{B}$ that defines the "core" of the distributions corresponding to c_1 and c_2. This argument shows that if any efficient adversary A can effectively distinguish c_1 from c_2 when given a single value from c_1 or c_2 such that $P(b) = true$, then there exists an efficient adversary A' that can effectively distinguish c_1 from c_2 when given (polynomially) many samples from one of the distributions. An additional condition required for this fact to hold is that the total probability mass of the core is not too small. The statement of this argument is shown in Listing 55, where `k1` and `k2` represent the probability mass of the core of `c1` and `c2`, respectively.

The proof of this fact is intuitive, and is illustrated in Figure 6.3. If the core of the distribution is sufficiently large, and if enough samples are taken from the distribution, then it is likely that at least one of these samples will fall within the core of the distribution. The constructed adversary A'

```
Definition RepeatCore_G(c : A -> Comp B) :=
  [a, s_A] <-$2 A1;
  b <-$ Repeat (c a) P;
  A2 s_A b.

Definition RepeatCore_Adv :=
  | Pr[RepeatCore_G c1] - Pr[RepeatCore_G c2] |.

Theorem DistMult_impl_RepeatCore :
  RepeatCore_Adv P c1 c2 A1 A2 <=
    DistMult_Adv c1 c2 A1 DM_RC_B2 n +
    (1 - k1)^n + (1 - k2)^n.
```

Listing 55: The Many to Core Theoreom

samples the distribution n times and gives the first "hit" in the core of the distribution to A which it uses to determine the source of the sample. When a hit is obtained, the distribution observed by A is identical to the distribution in which only the core is sampled. These distributions only differ when no hit is obtained after n attempts, but this event has negligible probability in n.

Full T-Set Security

I obtain security of the full T-Set scheme by combining the arguments in the previous sections. In order to apply the "Many to Core" argument, it must be shown that there is some positive $k \in \mathbb{Q}$, and the probability of successfully creating a T-Set from a database supplied by the adversary is at least k. This argument also requires that the simulator succeeds in one trial with probability at least k. Because these facts depend on the choice of parameters B and S, and we leave them as assumptions in the proof.

By combining the *Single-Trial T-Set* security proof with the assumptions related to k described in the previous paragraph, and with the arguments presented in Sections 6.3.1 and 6.3.1, I get the final security result in Listing 56. This theorem has the same assumptions as the "Single-Trial" security theorem in Listing 53, and the bounds of that theorem are present in this one.

```
Theorem TSet_secure :
  (forall i, PRF_NA_Advantage
    ({0,1}^lambda) (RndF_Range) F
    (TSet_IPRF_A1 i) TSet_IPRF_A2 <= F_Adv) ->
  PRF_NA_Advantage
    ({0,1}^lambda) ({0,1}^lambda) F_bar
    TSet_PRF_A1 TSet_PRF_A2 <= F_bar_Adv ->
  TSetAdvantage TSetSetup TSetGetTag
    L_T TSet_Sim A1 A2
  <= lambda * (F_bar_Adv + maxKeywords * F_Adv)
    + 2 * (1 - k)^lambda
```

Listing 56: T-Set Security

6.3.2 T-Set Correctness

The T-Set correctness proof has very similar structure to the security proof. The primary difference is that the ultimate goal is an inequality, rather than a proof that two values are "close." The proof uses slightly different forms of the "One to Many" and "Many to Core" arguments, and there are some interesting differences in the "single-trial" proof, which I highlight in this section.

SINGLE-TRIAL T-SET CORRECTNESS

The single-trial T-Set security proof was simplified by the fact that security is obvious when initialization fails. The empty T-Set resulting from an initialization failure clearly has no information that the adversary could use to distinguish it from the simulator. This argument is not so simple in the case of correctness, because an empty T-Set is obviously *not* correct. So I instead prove that the single-trial construction is conditionally correct. That is, a database and list of queries produced by the adversary is highly unlikely to result in a T-Set *on the first initialization attempt* that will produce an incorrect answer when queried. In the formalization of this definition (Listing 57), `good` is a predicate that indicates whether the TSetSetup routine produced a valid T-Set.

Notice that AdvCor_C_G unifies with the real game in the T-Set security definition (Listing 43). Since this definition is used in the single-trial T-Set security proof, I could use the result of

```
Definition AdvCor_C_G :=
  [t, q] <-$2 A;
  [tSet, k_T] <-$2 TSetSetup t;
  tags <-$ foreach (x in q)(TSetGetTag k_T x);
  t_w <- foreach (x in tags) (TSetRetrieve tSet x);
  t_w_correct <- foreach (x in q)
    (arrayLookupList _ t x);
  ret (good tSet && (t_w != t_w_correct)).

Definition AdvCor_C := Pr[AdvCor_C_G].
```

Listing 57: T-Set Conditional Correctness

this proof in the correctness proof to replace the game above with the ideal game from the security proof. Unfortunately, the simulator in the security proof eliminates some of the information required to show correctness. The security proof is a sequence of games, however, and I can use it to replace the game above with any game in that sequence. There is a game about halfway through in which many simplifications have been applied and the first PRF outputs are replaced with random values. So I save a significant amount of effort by reusing this result.

Next I perform a sequence of manipulations that simplify the T-Set and make it look more like the input database. For example, I put the values in the buckets in the same order as the input list rather than in a random order, I store and retrieve actual values instead of encryptions of values, and I make the structure one-dimensional. Then I replace the remaining PRF with a random function and replace the outputs with random values. Finally, I show that the T-Set is correct as long as there are no collisions in these random values, and I derive an expression for the probability of such a collision.

The sequence of games is diagrammed in Figure 6.4. The proof uses several of the same forms of equivalence from the security proof, and only the new labels are described in this paragraph. The equivalence labeled M uses the part of the security proof surrounded by a box in Figure 6.2 as an argument. In inequalities labeled C, I modify the game so that the adversary can also win by finding a collision during some operation. That is, the adversary can win by getting the game to produce a collision, or by satisfying the original "win" condition when there is no such collision. This allows a form of "identical until bad" reasoning for inequalities in which I can assume that there are no collisions going forward, and I will calculate the probability of collision and add it to the bounds in a later stage of the proof. E represents an equivalence by functional injection, in which I replace some operation on the outputs of an injective function with a related operation on the inputs of the function. These equivalences may use the assumptions provided by C steps, because if no collisions are encountered while interacting with a function, then that function behaves like an injection. In the final N equivalence of the correctness proof, I convert a simple collision-finding game into the corresponding probability expression B. The expression B is negligible in λ, and the bound on the advantage of the adversary in this theorem is the sum of B and the PRF advantage terms introduced by the $\approx$ equivalences.

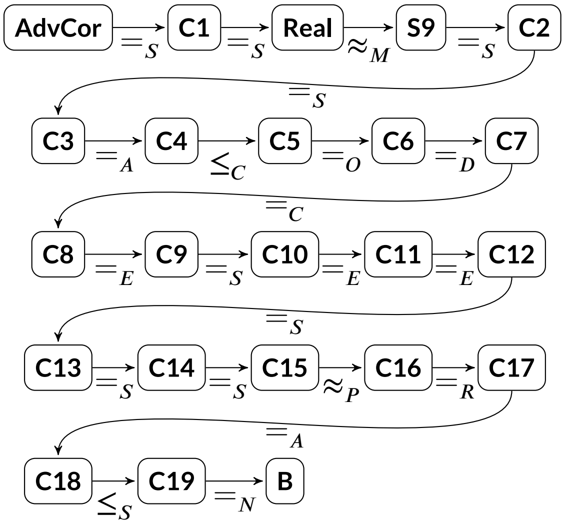


Figure 6.4: Single-Trial T-Set Correctness Games

The single-trial conditional correctness result is in Listing 58. In this listing, `maxMatches` is the maximum number of records matching any query, and `maxKeywords` is the maximum number of keywords in the database and queries supplied by the adversary. This result is similar to the single-trial security result because both proofs assume the functions `F` and `F_bar` are PRFs, and F is used as an *iterated* PRF in both proofs. The first term in the bounds of this theorem corresponds with B—the probability of a collision that would cause the result to be incorrect.

```
Theorem TSet_Correct_once :
  (forall i, PRF_NA_Advantage
    ({0,1}^lambda) RndF_R F
    (PRFI_A1 i) (PRFI_A2) <= F_Adv) ->
  PRF_NA_Advantage
    ({0,1}^lambda) ({0,1}^lambda) F_bar
    PRF_A1 PRF_A2 <= F_bar_Adv ->
  AdvCor_C TSetSetup_once TSetGetTag
    TSetRetrieve A1 A2 <=
  (maxKeywords * (S maxMatches))^2 / 2 ^ lambda
  + maxKeywords * F_Adv + F_bar_Adv.
```

Listing 58: Single-Trial T-Set Conditional Correctness

One to Many to Core Arguments

The "One to Many" and "Many to Core" arguments are slightly different from the ones used in the security proof. Rather than showing that the distance between two events is small, I only need to show that the probability of some event is small under the assumption that the probability of some other event is small. The required arguments are shown in Listing 59.

6.3.3 Full T-Set Correctness

The full T-Set correctness theorem is shown in Listing 60. This result is produced in a similar manner to the security result—the single-trial result is combined with the "One to Many" and "Many to Core" arguments along with some additional assumptions, and the single-trial bound appears in the bound of the full T-Set result. This proof also assumes a value `k` representing the probability that the `TSetSetup` routine succeeds in any attempt.

```
Definition TrueSingle_G :=
  a <-$ A1; b <-$ c a; ret (Q b).

Definition TrueMult_G :=
  a <-$ A1;
  bs <-$ foreach (x in (forNats n)) (c a);
  ret (fold_left (fun b x => b || (Q x)) bs false).

Definition TrueRepeat_G :=
  a <-$ A1; b <-$ Repeat (c a) P; ret (Q b).

Theorem TrueSingle_impl_Mult :
  Pr[TrueMult_G n] <= n * Pr[TrueSingle_G].

Theorem TrueMult_impl_Repeat :
  Pr[TrueRepeat_G] <=
  Pr[TrueMult_G n] + (k ^ n).
```

Listing 59: One to Many to Core Inequality Arguments

```
Theorem TSet_Correct :
  (forall i, PRF_NA_Advantage
    ({0,1}^lambda) RndF_R F
    (PRFI_A1 i) (PRFI_A2) <= F_Adv) ->
  PRF_NA_Advantage
    ({0,1}^lambda) ({0,1}^lambda) F_bar
    PRF_A1 PRF_A2 <= F_bar_Adv ->
  AdvCor TSetSetup TSetGetTag TSetRetrieve A1 A2 <=
  (1 - k)^lambda + lambda *
    ((maxKeywords * (S maxMatches))^2 / 2 ^ lambda
    + maxKeywords * F_Adv + F_bar_Adv).
```

Listing 60: T-Set Correctness

6.4 Proof Engineering

This proof was completed in approximately 6 months by a person with expert-level knowledge of FCF and moderate knowledge of the SSE scheme in question. Most of this time was spent in the "single-trial" security and correctness proofs. Table 6.1 provides the number of lines of Coq code and the number of intermediate games for each proof. To determine the number of intermediate games, I count only those games that would be produced by a cryptographer when developing the structure of the proof. In many cases, a high-level transformation is divided into several smaller transformations, each with its own intermediate game. The games used in these smaller transformations are not counted in the total number of games or to the lines of definition, but they do contribute to the number of lines of proof. The "Supporting Arguments" line measures only the arguments described in Sections 6.3.1, 6.3.1, and 6.3.2. This proof relies on a large amount of existing theory in the FCF library which comprises over 40,000 lines of Coq code, and this effort resulted in several thousand lines of additional reusable theory that was added to the standard library of FCF.

Table 6.1: Proof Complexity

Proof	Lines of Definition	Lines of Proof	Games
Single-Trial T-Set Security	447	3515	19
Single-Trial T-Set Correctness	611	5510	19
Supporting Arguments	48	1041	12
T-Set Security	0	1033	0
T-Set Correctness	0	998	0
SSE Scheme Security	257	920	8
Total	1363	13017	58

The table provides separate columns for definition (security definitions, constructions, intermediate games, constructed adversaries, and simulators) and proof (everything else including proof scripts, program logic judgments, and minor intermediate games). This separation proposes a division between the essential, cryptographic portion of the proof and the portion required by the mechanization. The division suggests that the mechanization increased the complexity of the proof by (roughly) a factor of 10. This increase in effort is large, but it should be considered reasonable when viewed in the context of the larger engineering effort of developing an implementation of this scheme. The proof is composed of several arguments, and the more complex arguments are further decomposed into a sequence of games. This decomposition provides ample opportunity to divide the proof development effort among a team of programmers.

It is important to note that this proof was completed in a largely manual style in which individual tactics are applied to transform the goal one step at a time. It is possible to adopt a more automated style in which Coq's tactic language (Ltac) is used to develop sophisticated tactics that discharge high-level goals. I could significantly reduce the number of lines of proof code by adopting this more automated style of proof. As an experiment, I re-developed the "SSE Scheme Security" proof using more automation. This is a relatively simple proof that is mostly structural and contains no interesting arguments, yet I was able to reduce the size of the proof by nearly 20 percent simply by making clever use of Ltac.

An important engineering concern is the extent to which artifacts developed for this proof could be reused in other proofs. Notably, the T-Set that was proved secure and correct in this proof is the same T-Set that is used in the more complex SSE schemes developed by Cash et al. By reusing the T-Set and its theory, I could greatly reduce the effort required to prove the security of any scheme that requires a correct or secure T-Set. Of course, the more general-purpose theory that was developed for this SSE proof could be directly reused by any proof.

Another consideration is the difficulty of changing the proof artifact to respond to changes in the scheme itself. First consider a minor change, such as a change to the representation (but not the content) of the database. I could address this change by proving that some game using the new database representation is equivalent to an existing game using the old representation. This change adds a new intermediate game to the sequence and increases the size of the proof. Another solution is to use a reduction to prove the security of the modified scheme assuming the security of the original scheme. This is a very powerful and general approach, but it also increases the size of the proof. A third option is to refactor the proof and change the database into an abstraction that could be instantiated with either representation. This solution may require more effort to implement, but it does not increase the size of the proof, and it results in a proof that is more tolerant of these changes in the future.

For more signification changes, it may be very hard to modify the proof. For example, if I wanted to prove adaptive security of the SSE scheme, I would need to change the way the scheme and the adversaries are modeled, add a random oracle, and change many of the security definitions to the appropriate adaptive security forms. This is a completely different proof, and none of the artifacts from the non-adaptive proof would be reused. However, much of the general-purpose theory in FCF that was developed for the non-adaptive security proof would still be applicable in the adaptive security proof.

6.5 Related Work

There has been a large amount of work in the area of formalizing cryptographic proofs in the last decade, but much of this work only involves simple examples used to demonstrate a tool, framework, or proof technique. This section focuses on mechanized proofs in the computational model related to non-trivial or practical constructions.

Several complex proofs have been completed in EasyCrypt and CertiPriv[12], a related system for reasoning about differential privacy. Stoughton[44] proved the security of a simplified version of a private information retrieval protocol. This is a fairly complex three-party protocol, but the simplified scheme only allows a query to retrieve the number of occurrences of a certain keyword in the database, and not the values associated with that keyword. Barthe et al.[7] demonstrate a formalization of differential privacy and a verification of a non-trivial smart metering system as an example. Almeida et al.[4] prove the security of a standardized public key encryption scheme. Barthe et al.[10] proved security of OAEP in CertiCrypt. Though this is a relatively simple construction, the proof of security is quite complex, comprising over 10,000 lines of Coq code.

Bhargavan et al.[18] verify an implementation of TLS using the F7 refinement type system. This is a remarkably complex proof, but several steps of the proof must be verified by hand due to the fact that F7 does not support reasoning about non-zero statistical distance between distributions. Barthe et al.[8] show how a variant of F* (a successor to F7) can be used to verify implementations of cryptographic schemes. This work provides several non-trivial examples including a certified privacy-preserving system for smart metering.

A certified proof of SSH[23] was completed in CryptoVerif, though this proof is limited to the transport layer protocol, and to the secrecy and authenticity of the session key only. This security does not extend to the messages sent over the channel due to a vulnerability in SSH. CryptoVerif was also used to formally verify the Kerberos network authentication system[21].

Roy et al.[42] use Protocol Composition Logic to verify the security of Diffie-Hellman key exchange as used in Kerberos and IPSec key management. Both are standardized protocols, and the models and formal proofs are quite complex.

6.6 Conclusion

In this chapter, I showed how FCF can be used to construct a proof of security for a complex cryptographic scheme. This result demonstrates that FCF is both scalable and flexible. In particular, the basic proof automation features provided by Coq are sufficient, and the higher-order abstraction available in Coq is very useful for proof engineering. In Chapter 7, I describe how FCF can be used to prove the correctness of implementations of cryptographic schemes in addition to models. Chapter 7 also includes a simple proof of security of HMAC that is used as part of a larger proof related to an implementation of HMAC written in C.

7

Provably Secure Implementations

Previous chapters have described efforts to prove the security of *models* of cryptographic systems. By verifying these models, it is possible to rule out significant categories of vulnerabilities. But many vulnerabilities are caused by issues that are outside of the model, or simply by errors in implementation. The ultimate goal of security verification is the verification of the *implementations* of cryptographic systems. Of course, the implementations are much more complex than the models, and research in this area is still in its initial stages.

In this chapter, I describe two mechanisms to ensure the security of cryptographic software. The first approach uses Coq's extraction mechanism to produce an implementation from an FCF model. The second approach uses the Verified Software Toolchain (VST) to show that source code written in C has certain cryptographic properties.

7.1 Extracting Code from FCF Models

In Section 4.5 I described an operational semantics that can be used to reason about the behavior of FCF computations on a traditional computer. This semantics is specified in a manner that makes it executable. Given a computation and a list of "random" input bits, I can run this computation to obtain either a value or an indication that the input bits were exhausted. I can use the `eval` command in Coq to run a computation in this manner, or I can extract the program as described in the remainder of this section.

Coq has an extraction mechanism that takes a Coq function and produces an equivalent Caml function. This extraction mechanism will also recursively extract all of the other functions and types required to execute the function. Given this extraction mechanism, I can produce executable code using the following process:

1. Extract both the operational semantics and the computation(s) of interest
2. Provide concrete instantiations for all abstract types and functions
3. Produce (or locate) boilerplate code that runs a computation and produces a result

The last step in this sequence is necessary because the operational semantics only describes how a computation takes a single step. Because all Coq functions must terminate, I cannot write a function in Coq that repeatedly causes the computation to take a step under the operational semantics until it (possibly) terminates. So I must provide this code in Caml. This code can also obtain random bits and provide them to the semantics when needed. Listing 61 contains an example program that runs a computation. In this listing, `evalDet_step` is the function that defines the operational semantics, and `randomBits` is a function that uses `Random.Bool` to obtain a number of random bits from the environment when needed.

```
let rec runComp_h c s =
  match (evalDet_step c s) with
  | Cs_done (b, s’) -> Cs_done (b, s’)
  | Cs_eof -> let newBits = randomBits 1000 in
        runComp_h c (append s newBits)
  | Cs_more (c’, s’) -> runComp_h c’ s’

exception InvalidCompState;;

let runComp c =
  match (runComp_h c Nil) with
  | Cs_done (b, s’) -> b
  | Cs_eof -> raise InvalidCompState
  | Cs_more (c’, s’) -> raise InvalidCompState
```

Listing 61: Boilerplate Code that Runs a Computation

To demonstrate that this approach produces working code, I extracted the PRF encryption scheme described in Section 5.2. I used the Caml code in Listing 61 to run the computation, and I provided a small number of additional functions to convert between standard Caml types (e.g. Boolean and integer) and the extracted types. I instantiated the “PRF” with the *xor* function for bit vectors. Obviously, *xor* is not a PRF, but this simple function allows me to test the extraction mechanism and verify that I can run the extracted code. If I replace this function with a function that is believed to be a PRF, then the resulting code would have the security properties guaranteed by the proof in Section 5.2.

It’s important to note that the extracted program is not very efficient. It is written in Caml and can be compiled or interpreted under OCaml. Even when compiled, the resulting OCaml program is likely to be less efficient than an equivalent C program, and the garbage collection of OCaml can be problematic in real-time systems. A more significant issue for efficiency is that the resulting program uses a number of Coq types and operations (e.g. unary natural numbers and their related operations) which were developed for ease of modeling and reasoning instead of efficiency.

The extracted code is probably too inefficient to be used in production, but it is still valuable. It can be used to develop a prototype in a "proof of concept" stage of development. That is, a new cryptosystem can be modeled and proved correct in FCF, and some basic testing can be performed on the extracted implementation. This implementation would be replaced by a more efficient implementation at a later stage. The extracted code could also be used as a reference implementation for testing purposes. When testing the production implementation, the output could be compared to that of the extracted reference implementation in order to find bugs and vulnerabilities.

7.2 Verifying C Code

By combining FCF with additional systems for reasoning about C code, it is possible to obtain a fully verified implementation of a cryptographic system that is efficient and can be used in production. In this section, I describe an approach used to verify the cryptographic properties of an implementation of HMAC[14] written in C. This section describes joint work with Andrew Appel, Lennart Beringer, and Katherine Ye, and my main contribution is a model of HMAC and a proof of its cryptographic properties.

7.2.1 HMAC

HMAC is a symmetric message authentication code (MAC) scheme based on a secure hash function. It can be used to establish the authenticity of messages sent between two parties that share a common symmetric key. For example, if Alice wants to send a message M to Bob, she can send the pair $(M, HMAC(K, M))$ where K is the key shared by Alice and Bob. When Bob receives this pair, he can check that the second value equals $HMAC(K, M)$ to verify that the message came from Alice (or someone who knows *K*) and it has not been modified. In order for such a MAC function to be secure, it must be the case that an adversary who does not know *K* would have great difficulty producing some message *M'* and a forged MAC value *Z* such that *Z* = *HMAC(M', K)*. If HMAC is a PRF, then this unforgeability is implied, and I will prove that our implementation of HMAC is a PRF.

7.2.2 VERIFIED SOFTWARE TOOLCHAIN

We use the Verified Software Toolchain[5] (VST) to reason about C code and its corresponding machine code. VST is a Coq library that provides a separation logic for C that allows us to prove that a program has some specification in the form of a precondition and a postcondition. Notably, we can use VST to prove that some C code has the same input/output behavior as a Coq function. So given a Coq function that specifies the behavior of HMAC, we can prove that some C code is functionally equivalent to that Coq function.

VST is built on top of CompCert[36], which is a fully-verified compiler for C programs. CompCert provides a semantics for C and a semantics for machine code, and a mechanized proof establishes that the machine code that results from compilation has the same behavior as the input C program. Therefore, VST can be used to prove that an implementation in machine code has certain correctness or security properties.

7.2.3 MECHANIZED SECURITY AND CORRECTNESS OF HMAC

We focus on the implementation of HMAC provided in OpenSSL version 0.9.1c, and we prove the following:

1. The HMAC code behaves identically to a formalization of the FIPS 198-1 Keyed-Hash Message Authentication Code specification. The implementation of SHA-256 used as the un-

derlying hash function behaves identically to a formalization of the FIPS 180-4 Secure Hash Standard.

2. An abstract specification of HMAC is a PRF given certain (reasonable) cryptographic assumptions on the underlying hash function.

3. FIPS 198-1, when using FIPS 180-4 as the underlying hash function, is a refinement of the abstract HMAC specification.

Because the PRF property is preserved by functional equivalence and refinement, we obtain the following machine-checked theorem.

Theorem 13. The assembly-language program that results from compiling OpenSSL 0.9.1c using CompCert implements the FIPS standards for HMAC and SHA-256, and implements a cryptographically secure PRF subject to certain cryptographic assumptions about SHA-256 (enumerated in Section 7.2.5).

My contribution to this result is the abstract specification for HMAC and the proof of its cryptographic properties. I will describe this contribution in the remainder of this section and omit details of other portions of the proof.

7.2.4 Cryptographic properties of HMAC

This subsection describes a mechanization of a cryptographic proof of security of HMAC. The final result of this proof is similar to the first HMAC proof of Bellare et al.[14], though the structure of the proof and some of the definitions are influenced by Bellare's 2006 proof[13]. This proof uses a somewhat abstract model of HMAC in which keys are in $\{0,1\}^b$ (the set of bit vectors of length b), inputs are in $\{0,1\}^*$ (bit lists), and outputs are in $\{0,1\}^c$ for arbitrary b and c s.t. $c \leq b$. An implementation of HMAC would require that b and c are multiples of some word size, and the input is an array of words, but these issues are typically not considered in cryptographic proofs.

In order to use security results related to this specification, we must show that this specification is appropriately related to the FIPS 198-1 HMAC specification. I chose to prove the security of the abstract specification, rather than directly proving the security of the FIPS specification, because there is significant value in this organization. Primarily, this organization allows me to use the exact definitions and assumptions from the cryptography literature, and I therefore gain greater assurance that the definitions are correct and the assumptions are reasonable. Also, this approach demonstrates how an existing mechanized proof of cryptographic security can be used in a verification of the security of an implementation. This organization also helps decompose the proof, and it allows me to deal with issues of cryptographic security in isolation from issues related to implementation.

7.2.5 HMAC SECURITY

I mechanized a proof of the following fact. If h is a compression function, and h^* is a Merkle-Damgård[38,26] hash function constructed from h, then HMAC based on h^* is a pseudorandom function (PRF) assuming:

1. h is a PRF.
2. h^* is weakly collision-resistant (WCR).
3. The dual family of h (denoted $\bar{h}$) is a PRF against $\oplus$-related-key attacks.

The formal definition of a PRF is shown in Listing 62. In this definition, `f` is a function in `K -> D -> R` that should be a PRF. The adversary `A` is an `OracleComp` that interacts with either an oracle constructed from `f` or with `randomFunc`, a random function constructed by producing random values

```
Definition f_oracle(k : K)(x : unit)(d : D) :=
  ret (f k d, tt).

Definition PRF_G0 : Comp bool :=
  k <-$ RndKey;
  [b, _] <-$2 A (f_oracle k) tt; ret b.

Definition PRF_G1 : Comp bool :=
  [b, _] <-$2 A (randomFunc) nil; ret b.

Definition PRF_Advantage :=
  | Pr[PRF_G0] - Pr[PRF_G1] |.
```

Listing 62: Definition of a PRF

for outputs and memoizing them so they can be repeated the next time the same input is provided. The `randomFunc` oracle uses a list of pairs as its state, so an empty list is provided as its initial state.

This security definition is provided in the form of a game in which the adversary tries to determine whether the oracle is `f` (in game 0) or a random function (in game 1). After interacting with the oracle, the adversary produces a Boolean value, and the adversary wins if this value is likely to be different in the games. I define the *advantage* of the adversary to be the difference between the probability that it produces "true" in game 0 and in game 1. I can conclude that `f` is a PRF if this advantage is sufficiently small.

The definition of a weakly collision-resistant function is shown in Listing 63. This definition uses a single game in which the adversary is allowed to interact with an oracle defined by a keyed function `f`. At the end of this interaction, the adversary attempts to produce a collision—a pair of different input values that produce the same output. In this game, I use `?=` and `!=` to mean tests for equality and inequality, respectively. The advantage of the adversary is the probability with which it is able to locate a collision.

Finally, the security proof assumes that a certain keyed function is a PRF against $\oplus$-related-key attacks (RKA). This definition (Listing 64) is similar to the definition of a PRF, except the adversary is also allowed to provide a value that will be `xor`ed with some fixed value to produce the key used by the PRF. Note that this assumption is on the *dual family* of h, in which the roles of inputs and

```
Definition Adv_WCR_G :=
  k <-$ RndKey;
  [d1, d2, _] <-$3 A (f_oracle k) tt;
  ret ((d1 != d2) && ((f k d1) ?= (f k d2))).

Definition Adv_WCR := Pr[Adv_WCR_G].
```

Listing 63: Definition of Weak Collision-Resistance

keys are reversed. So a single input value is chosen at random and fixed, and the adversary queries the oracle by providing values which are used as keys.

```
Definition RKA_F s p :=
  ret (f ((fst p) xor k) (snd p), tt).
Definition RKA_R s p :=
  randomFunc s ((fst p) xor k, (snd p))

Definition RKA_G0 :=
  k <-$ RndKey; [b, _] <-$\$$2 A RKA_F tt; ret b.

Definition RKA_G1 :=
  k <-$ RndKey; [b, _] <-$\$$2 A RKA_R nil; ret b.

Definition RKA_Advantage :=
  | Pr[RKA_G0] - Pr[RKA_G1] |.
```

Listing 64: Definition of Security against $\oplus$ Related-Key Attacks

The proof of security has the same basic structure (Figure 7.1) as Bellare's 2006 HMAC proof[13], though I simplify the proof significantly by assuming h^* is WCR. The proof makes use of a nested MAC (NMAC) construction that is similar to HMAC, but it uses h^* in a way that is not typically possible in implementations of hash functions. The proof begins by showing that NMAC is a PRF given that h is a PRF and h^* is WCR. Then I show that NMAC and HMAC are "close" (that no adversary can effectively distinguish them) under the assumption that $\bar{h}$ is a $\oplus$-RKA-secure PRF. Finally, I combine these two results to derive that HMAC is a PRF.

I also mirror Bellare's proof by reasoning about slightly generalized forms of HMAC and NMAC (called GHMAC and GNMAC) that require the input to be a list of bit vectors of length b. The proof also makes use of a "two-key" version of HMAC that uses a bit vector of length $2b$ as the key.

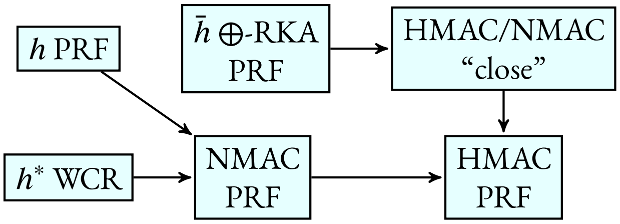


Figure 7.1: HMAC Security Proof Structure

To simplify the development of this proof, I build HMAC on top of these intermediate constructions in the abstract specification (Listing 65).

```
Definition h_star k (m : list (Bvector b))
  := fold_left h m k.
Definition hash_words := h_star iv.

Definition GNMAC k m :=
  let (k_Out, k_In) := splitVector c c k in
  h k_Out (app_fpad (h_star k_In m)).

Definition GHMAC_2K k m :=
  let (k_Out, k_In) := splitVector b b k in
  let h_in := (hash_words (k_In :: m)) in
  hash_words (k_Out :: (app_fpad h_in) :: nil).

Definition HMAC_2K k (m : list bool) :=
  GHMAC_2K k (splitAndPad m).

Definition HMAC (k : Bvector b) :=
  HMAC_2K ((k xor opad) ++ (k xor ipad)).
```

Listing 65: HMAC Abstract Specification

In Listing 65, `splitAndPad` is a function that produces a list of bit vectors from a list of bits (padding the last bit vector as needed), and `app_fpad` is a padding function that produces a bit vector of length b from a bit vector of length c. In the definition of the `HMAC` function, we use constants `opad` and `ipad` to produce a key of length $2b$ from a key of length b. These functions and constants are parameters to the definitions, and concrete values for these items are provided by the FIPS specifications.

The statement of security for HMAC is shown in Listing 66. We show that HMAC is a PRF by giving an expression that bounds the advantage of an arbitrary adversary A. This expression is the sum of three terms, where each term represents the advantage of some adversary against some other security definition.

The listing describes all the parameters to each of the security definitions. In all these definitions, the first parameter is the computation that produces random keys, and in `PRF_Advantage` and `RKA_Advantage`, the second parameter is the computation that produces random values in the range of the function. In all definitions, the penultimate parameter is the function of interest, and the final parameter is some constructed adversary. The descriptions of these adversaries are omitted for brevity, but only their computational complexity is relevant (e.g. all adversaries are in ZPP assuming adversary `A` is in ZPP).

```
Theorem HMAC_PRF:
  PRF_Advantage ({0, 1}^b) ({0, 1}^c) HMAC A <=
  PRF_Advantage ({0, 1}^c) ({0, 1}^c) h B1 +
  Adv_WCR ({0, 1}^c) h_star B2 +
  RKA_Advantage ({0, 1}^b) ({0, 1}^c)
    (BVxor b) (dual_f h) B3.
```

Listing 66: Statement of Security for HMAC

It is possible to view the result in Listing 66 in the asymptotic setting, in which there is a security parameter η, and parameters c and b are polynomial in η. In this setting, it is possible to conclude that the advantage of A against HMAC is negligible in η assuming that each of the other three terms is negligible in η. I can also view this result in the *concrete setting*, and use this expression to obtain *exact security* measures for HMAC when the values of b and c are fixed according the sizes used by the implementation. The latter interpretation is more informative, and probably more appropriate for reasoning about the cryptographic security of an implementation.

7.3 Related Work

The result described in Section 7.2.1 is the first fully foundational end-to-end verification of the cryptographic properties of a machine code implementation. Some previous efforts have produced similar results that are more limited or contain gaps in the mechanization that must be verified manually. This section describes efforts related to verifying cryptographic security (in the computational model) of implementations.

EasyCrypt has been used in a proof of security of an implementation of OAEP with RSA[4]. The implementation is obtained by converting a program in the language of EasyCrypt to C. This C program is compiled to machine code using CompCert, and a separate tool verifies that machine code leaks no more information in the program counter trace than the C program. This mechanization contains several gaps that require inspection. The program that extracts the C program is unverified Python code, and there is no guarantee that the extracted program is equivalent to the EasyCrypt program. Further, there is no formal relationship between the semantics of C and the semantics of EasyCrypt, so it is necessary to inspect these semantics to ensure that the security properties of an EasyCrypt program transfer to the corresponding C program.

Cadé and Blanchet showed[24] how to extract a Caml program from a CryptoVerif model. The result is accompanied by a proof that the extraction mechanism is correct and the extracted code enjoys the same security properties of the model. This proof is not mechanized, however, and it is necessary to trust that the extraction is implemented correctly. Aizatulin et al.[3] developed a system to extract a CryptoVerif model from C code. This is a very useful and practical system, but there is no mechanized proof that this extraction produces a CryptoVerif program that is semantically equivalent to the C program.

Bhargavan *et al.*[19] prove the security of a implementation of TLS in F# using the F7 type system. This is a remarkably complex proof, and the resulting code is a fully-feature reference implementation. F7 is not capable of probabilistic reasoning, however, and many parts of the proof are left as assumptions.

7.4 Conclusion

In this chapter, I described two different mechanism for reasoning about the security of cryptographic implementations using FCF. These proofs were enabled by the flexibility of FCF and direct integration with Coq, which allow results in FCF to be easily combined with other Coq mechanisms, libraries, and proofs.

8

Summary and Conclusion

I have presented a new framework for mechanized cryptographic proofs which improves on the state of the art in several areas, while making acceptable sacrifices in others. Notably, FCF features a fully foundational design (Chapters 3 and 4) that supports trustworthy extension, and it provides sufficient ease of use to allow the development and checking of complex proofs (Chapter 6). FCF also supports advances in the state of the art of verification of cryptographic implementations (Chapter 7) by providing a mechanism to combine a proof of cryptographic security with a proof of functional correctness in Coq.

I repeat the comparison table from Chapter 3 in Table 8.1. The scores in this table are explained in Chapter 3, and I have provided justification for the scores of FCF throughout this paper. FCF performs relatively well for all attributes except for *Automation*, though I have shown in Chapter 6 that the automation and other features provided by Coq and FCF support large proofs of security for complex schemes.

	FCF	EasyCrypt	CertiCrypt	CryptoVerif	F*
Familiarity	4	4	2	4	2
Automation	2	3	2	5	3
Trustworthiness	5	4	5	4	3
Expressivity	4	5	5	2	3
Extensibility	5	3	4	2	3
Concrete Security	5	5	5	5	2
Abstraction	5	4	4	2	2
Implementation	5	4	1	4	4

Table 8.1: Comparison of Mechanized Cryptography Systems

8.1 Choosing a Cryptographic Proof Framework

All of the systems described in Table 8.1 are very capable systems for developing and checking cryptographic proofs. When deciding on a system to use to mechanize a proof, the correct choice will largely be determined by the details of the cryptographic scheme and the desired outcome of the proof.

If CryptoVerif is capable of modeling the cryptographic scheme of interest and the security definitions, then using this tool would probably be a wise choice. The level of automation in CryptoVerif significantly reduces the level of effort required to complete the proof. Unfortunately, CryptoVerif is not capable of expressing many interesting cryptographic schemes and security definitions.

If the goal is a proof related to an implementation, or the level of rigor required in the proof is relatively low, then perhaps the proof should be completed using F*. The lack of probabilistic reasoning results in more "gaps" in the proof compared to the other framework, but the effect of these gaps can be reduced by properly engineering the proof. Overall, F* strikes a good balance between ease of use and level of rigor, and the fact that the F# code that defines the scheme is executable is a significant benefit.

The choice between FCF, CertiCrypt, and EasyCrypt probably comes down to particular details of the proof and personal preferences of the developer. If the developer is comfortable with Coq, then it may be more reasonable to complete the proof in FCF or CertiCrypt. If not, EasyCrypt may be a better choice because the tool is simpler and easier to learn than Coq. If EasyCrypt lacks the theory required to complete the proof, and the developer is not comfortable modifying the EasyCrypt source code to add this theory, then FCF or CertiCrypt would be a better choice. There may be certain constructions or definitions that are difficult to model in FCF due to its pure functional language that is not Turing-complete. In this case, CertiCrypt or EasyCrypt may be a better choice.

In summary, none of these systems are clearly better in all circumstances, and the relative advantages of these systems are limited to certain categories of circumstances. Choosing the most appropriate system for a particular proof requires a good understanding the subtleties of the proof as well as the capabilities of these systems.

8.2 Future Work

Though the last decade has produced a significant amount of improvement to the state of the art in mechanization of cryptographic proofs, this technology still has a long way to go before it can be routinely used by cryptographers. In the remainder of this chapter, I will describe the main weaknesses in this technology and propose avenues for future research.

A significant issue with current general-purpose cryptographic proof systems is that they require the developer to reason about the cryptographic scheme at a very low level of abstraction. For example, where a conventional proof would say "by a one-time pad argument, the values of *x* are uniformly distributed." In a mechanized proof, several steps are required to demonstrate that the one-time pad argument can be applied to the current game, indicate where it should be applied, and transform the game into the desired final form. This process may produce proof obligations related to program equivalence or similar goals that require the develop to produce loop invariants or prove other judgments on programs. Of course, the one-time pad argument is a very simple one, and this issue is only magnified when more complex arguments are applied.

The solution to this problem is to develop a higher-level interactive proof system that allows the developer to select an argument and indicate an expression or other location in the game where that argument should be applied. Proof search could be used to locate a proof that the argument is applicable at that location, and heuristics could even be used to propose candidate locations where an argument might be valid. When necessary, the developer will be prompted for loop invariants or other facts that are needed by the proof search. The system should search judgments that have been proven in the past, since the same (or related) judgments are often reused in different parts of the proof. This system can simply be a front end to FCF or EasyCrypt, so it does not need to be fully trusted.

Another issue with current cryptographic proof frameworks is that they all lack a good, general-purpose mechanism for reasoning about the efficiency and complexity of programs. CertiCrypt and CryptoVerif include mechanisms that ensure all programs are probabilistic polynomial time, but this approach does not support other cost models and complexity classes. FCF supports any cost model and complexity class, but only a simple demonstration using an axiomatic cost model has been provided so far. This problem will always be challenging since these frameworks are extensible. It is often necessary to assign a cost to an abstract function that only has an axiomatic definition, and so the cost of the function must be assigned axiomatically.

More work is necessary to demonstrate that axiomatization of cost models is sufficiently expressive and provides a reasonable level of assurance. For example, it would be informative to develop a uniform polynomial time cost model for FCF. Another approach is to develop a separate programming language and/or semantics for each cost model of interest, and program the constructed adversaries (and other programs) of interest in that language. This language should be sufficiently expressive to contain all of the necessary types and operations used by the constructed adversaries, and it should have a semantics that indicates the cost of running a program.

Finally, there is still much work to be done in the area of reasoning about cryptographic implementations. In Chapter 7, I describe the first fully foundational, end-to-end proof of the cryptographic security properties of an implementation, but this is still just initial work in this area. Future work should consider constructions that flip coins and use FCF's operational semantics to show that the result is equivalent to a C program that reads random data from a stream. The proof of adequacy of the denotational semantics assumes that this random input is uniform, but future work should consider the practical issue that the randomness supplied to a program is never truly uniform. In this case, it is important to bound the "insecurity" introduced by using input that is merely "close" to uniformly random.

Another issue with implementations is reasoning about side channels. The proof of OAEP in EasyCrypt[4] uses a separate analysis to ensure that the implementation does not leak information through side channels. A more general approach would include side channels in the cryptographic model, and the proof would assume restrictions on the information that is leaked to the adversary through these side channels. Then it may be possible to prove that an implementation leaks no more through side channels than what is assumed in the cryptographic proof.

A

Adequacy of Operational Semantics

In Section 4.5 I describe an operational semantics that can be used to reason about implementations of cryptographic systems and I state that this semantics is equivalent (in a particular sense) to the denotational semantics used to reason about cryptographic properties. The denotational semantics is adequate with respect to the operational semantics under a particular interpretation of probability. That is, the denotational semantics corresponds to the infinite unrolling of the small-step semantics when the input bits are assumed to be uniformly distributed. In this chapter, I describe this fact in greater detail, and I describe the Coq proof of this fact, which is interesting and non-trivial.

A.1 The Value of Adequacy

Similar frameworks for developing cryptographic proofs are based only on a probabilistic semantics, with no semantics that corresponds to a traditional model of computation. FCF includes a traditional operational semantics along with an equivalent probabilistic denotational semantics because several benefits are derived from this organization.

The primary value of the operational semantics and the proof of adequacy is that this fact enables FCF to reason about implementations of cryptographic schemes in a highly trustworthy manner. Implementations of cryptographic schemes behave in the manner of the operational semantics, in which values are stored in memory and random bits are obtained by reading from some list or stream provided by the environment. By proving that an implementation is equivalent to (or a refinement of) some model when executed under the operational semantics, it is possible to conclude that the implementation inherits the security properties of the model. More information about secure implementations is provided in Chapter 7.

A significant benefit of the proof of adequacy is that any cryptographic construction that is proven secure will also be secure when interpreted under the operational semantics. In conventional cryptographic proofs, procedures are modeled as probabilistic polynomial time Turing machines. Because the operational semantics provides a basis for a similar model of computation, and because conclusions are derived from a probabilistic semantics that is equivalent to that model, security claims in FCF system are very similar to the claims in conventional proofs in cryptography.

A related benefit is that it is not necessary to trust that the probabilistic semantics describes some reasonable behavior of a probabilistic programming language. Instead, one can inspect the operational semantics in order to conclude that it is reasonable, and also inspect the statement of adequacy. If the probabilistic semantics is not trusted, it can be changed it at will in order to support additional programming constructs and arguments.

Additionally, it is often necessary to prove that some program transformation is sound with respect to the probabilistic semantics, and it may be easier to prove that the transformation is sound with respect to the operational semantics. By proving these semantics equivalent, we can conclude that any two programs that are equivalent with respect to the operational semantics are also equivalent with respect to the denotational semantics. For example, equivalences related to loop unrolling are trivial to prove under the operational semantics, and much more challenging under the denotational semantics.

A.2 ADEQUACY THEOREM

Section 4.5 contained a statement of the theorem of adequacy, which is repeated in Theorem 14. In this section, I provide more information about the definitions that related to this theorem, and I described its proof. The proof itself is very interesting, and it contains several insights into proving facts related to discrete probability distributions and (infinite) limits in Coq.

Theorem 14. If c is well-formed, then $\lim\limits_{n \to \infty} [c]_n = [\![c]\!]$

A.2.1 WELL-FORMED COMPUTATIONS

It is possible to write non-terminating programs in FCF, such as the following repeated experiment:

`Repeat (ret 0) (fun x => x ?= 1).`

This program runs the command `(ret 0)` until the result is `1`, which of course will never happen. A program which does not terminate in all cases corresponds to a distribution in which the probability mass does not sum to one. We only want to consider probability distributions, so we will rule out such programs by requiring programs to be well-formed. A computation is well-formed if, for all Repeat statements in the computation, the support of the repeated computation contains at least one value that is accepted by the termination predicate. Note that a well-formed computation will not necessarily terminate in the operational semantics, but it will terminate with probability one when the input is a uniformly distributed stream of random bits.

The theorem of adequacy only applies to well-formed computations because the denotation of a non-well-formed computation is undefined. Recall the denotation of a Repeat statement:

$$[\![Repeat\ c\ P]\!] = \lambda x.(1_P\ x)\ ([\![c]\!]\ x) \left(\sum_{b \in P} ([\![c]\!]\ b) \right)^{-1}$$

The final term in this product is the inverse of the total probability mass that matches the predicate P. If the computation is not well-formed, then this sum is zero and the value of the inverse term is undefined.

A.2.2 Low Distribution Approximation

Given a program, I can approximate the probability that the program returns some value x as follows:

- Let L be the list of all possible bit lists of length n
- Run the computation (under the operational semantics) on all lists in L and collect the results in list R
- Let c be the number of results in R that equal *some x s'* for some s'
- The approximation at level n is $c/length(L) = c/2^n$

This approximation is “low” because some of the executions will produce *eof*, and these results are not included in the count. I use the notation $[c]_n$ to denote the low distribution approximation of computation c at level n.

A.2.3 PROOF OF ADEQUACY

In the remainder of this section I sketch the proof of adequacy of the probabilistic semantics. Like all other facts related to FCF, this fact has been formally proven in Coq, and the description is included in this paper only for the purpose of illustration.

The proof proceeds by induction on the structure of the computation c. The base cases (Ret and Rnd) can be discharged directly, whereas the inductive cases (Bind and Repeat) require a significant amount of explanation. We will use the case of Bind to explain the challenge with these cases.

In the case of Bind, the goal is:

$$\lim_{n \to \infty} [Bind\ c\ f]_n = [\![Bind\ c\ f]\!]$$

and I have the following induction hypotheses:

$$\lim_{n \to \infty} [c]_n = [\![c]\!]$$

$$\forall b \in supp([\![c]\!]), \lim_{n \to \infty} [f\ b]_n = [\![f\ b]\!]$$

These induction hypotheses tell me that the approximations are correct for the subterms. I need to use these induction hypotheses to reach the goal, but I cannot apply them directly. The problem is that each hypothesis considers an approximation at level n, but when I approximate the term "Bind c f" at level n, I don't use n bits for each subterm. Rather, I use $t \leq n$ bits for the first subterm, and then $t' \leq n - t$ bits for the second subterm.

The solution to this problem involves an alternative method of approximating distributions for Bind terms. This method, called the *bind approximation*, is provided in Definition 5.

Definition 5 (Bind Approximation).

$$B[c, f]_n = \lambda a. \sum_{b \in supp(\llbracket c \rrbracket)} ([c]_n\ b)\ \ ([f\ b]_n\ a)$$

The bind approximation has two important features. First, an approximation at level n uses up to n bits for each subterm, allowing me to use my induction hypotheses. Second, it is structurally the same as the denotation of a Bind term, except approximations of subterms are used instead of their denotations. I use the bind approximation to prove the limit of the low distribution approximation for bind terms using the squeeze theorem. That is, I show that there are two functions (both derived from the bind approximation) that bound the low distribution approximation from above and from below, and both these functions have the desired limit. The rest of this proof is described in Theorems 15, 16, 17, and 18.

Theorem 15 (Bounded from Above). For all n,

$$[Bind\ c\ f]_n \leq B[c, f]_n$$

Proof. The low distribution approximation only gets to use n bits total, whereas the bind approximation is allowed to use n bits per subterm. Clearly, the bind approximation must be at least as good as the low distribution approximation, so the probability of any event in the bind approximation must be greater than or equal to the probability of the same event in the low distribution approximation. □

Theorem 16 (Bounded from Below). For all n,

$$B[c, f]_{n/2} \leq [Bind\ c\ f]_n$$

Proof. Both approximations use at most n bits total, but $\boldsymbol{B}[c, f]_{n/2}$ may only use at most $n/2$ bits for each subterm. So for the cases in which c requires more than $n/2$ bits, the approximation produced by [Bind c f]$_n$ will be at least as good as the approximation produced by $\boldsymbol{B}[c, f]_{n/2}$. □

The formal proofs of Theorem 15 and 16 are much more complex than the informal proofs included in this paper. To conclude that some approximation is "at least as good" as some other approximation, I consider distribution approximations in the form of binary trees, where I branch on the value of each input bit, and I can compute the probability of some event by summing the leaves corresponding to that event and dividing by the total number of leaves. I developed additional alternative approximations that produce trees, and then proved that these tree-based approximations are identical to the corresponding non-tree-based approximations. To prove that some tree-based approximation *t* is at least as good as some other approximation *t'*, I show that the two trees are identical, except *t* is allowed to have an arbitrary tree any place where *t'* has a leaf node containing no value (corresponding to input list exhaustion). Once it is established that *t* is at least as good as *t'*, a simple proof by induction will show that the probability of any event in *t* is greater than or equal to the probability of the same event in *t'*.

I have shown that the low distribution approximation is bounded on both sides by these functions derived from the bind approximation. Now I show that the infinite limit of both of these functions is equal to the value given by the denotational semantics. Then, by the squeeze theorem, the infinite limit of the low distribution approximation for ***Bind*** is equal to the value given by the denotational semantics.

Theorem 17 (Limit of "Above" Function).

$$\lim_{n\to\infty} [c]_n = [\![c]\!] \wedge \forall b \in supp([\![c]\!]), \lim_{n\to\infty} [f\ b]_n = [\![f\ b]\!]$$

$$\Rightarrow \lim_{n\to\infty} B[c, f]_n = [\![(Bind\ c\ f)]\!]$$

Proof. After unfolding some definitions I get the following goal:

$$\begin{aligned}\forall a, \lim_{n\to\infty} \sum_{b\in supp(\llbracket c\rrbracket)} ([c]_n\ b) * ([f\ b]_n\ a)\\ = \sum_{b\in supp(\llbracket c\rrbracket)} \llbracket c\rrbracket b * (\llbracket(f\ b)\rrbracket a)\end{aligned}$$

By the (iterated) sum rule of limits, it is sufficient to show:

$$\begin{aligned}&\forall b \in supp(\llbracket c\rrbracket), \forall a,\\ &\lim_{n\to\infty} ([c]_n\ b) * ([f\ b]_n\ a) = \llbracket c\rrbracket b * (\llbracket(f\ b)\rrbracket a)\end{aligned}$$

This fact follows from our hypotheses and the product rule of limits. ☐

Theorem 18 (Limit of "Below" Function).

$$\begin{aligned}\lim_{n\to\infty}[c]_n = \llbracket c\rrbracket \wedge \forall b \in Sup(\llbracket c\rrbracket), \lim_{n\to\infty}[f\ b]_n = \llbracket f\ b\rrbracket\\ \Rightarrow \lim_{n\to\infty} B[c,f]_{n/2} = \llbracket(Bind\ c\ f)\rrbracket\end{aligned}$$

Proof. This statement is just like the statement of Theorem 17, except the approximation is taken at level $n/2$ instead of level n. Since we are considering limits at infinity, this fact clearly follows from Theorem 17. ☐

The proof for the Repeat case is very similar. I create an alternative approximation for Repeat, denoted $R[c,P]_n$, where c is the repeated experiment, P is the termination predicate, and n is the approximation level. This approximation acts as if the computation c is allowed to read n bits from the input sequence in each iteration. I then squeeze the actual distribution approximation function between $R[c,P]_{\lfloor\sqrt{n}\rfloor}$ and $R[c,P]_n$.

A.3 CONCLUSION

The proof of adequacy required a large amount of effort to complete, but the value is significant. Not only does this fact allow me to use either semantics as a foundation to complete proofs of security, it also supports proofs related to implementations using the operational semantics. Without this theorem it would be necessary to assume a relationship between the two semantics, making any result that uses this assumption less trustworthy.

References

[1] Abadi, M. & Rogaway, P. (2000). Reconciling two views of cryptography (the computational soundness of formal encryption). In *Proceedings of the International Conference IFIP on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics*, TCS '00 (pp. 3–22). London, UK, UK: Springer-Verlag.

[2] Affeldt, R., Tanaka, M., & Marti, N. (2007). Formal proof of provable security by game-playing in a proof assistant. In *Proceedings of the 1st international conference on Provable security*, ProvSec'07 (pp. 151–168). Berlin, Heidelberg: Springer-Verlag.

[3] Aizatulin, M., Gordon, A. D., & Jürjens, J. (2012). Computational verification of c protocol implementations by symbolic execution. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12 (pp. 712–723). New York, NY, USA: ACM.

[4] Almeida, J. B., Barbosa, M., Barthe, G., & Dupressoir, F. (2013). Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations. In *Proceedings of the 2013 ACM SIGSAC conference on Computer &#38; communications security*, CCS '13 (pp. 1217–1230). New York, NY, USA: ACM.

[5] Appel, A. W. (2011). Verified software toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ESOP'11/ETAPS'11 (pp. 1–17). Berlin, Heidelberg: Springer-Verlag.

[6] Backes, M. & Unruh, D. (2008). Computational soundness of symbolic zero-knowledge proofs against active attackers. In *21st IEEE Computer Security Foundations Symposium, CSF 2008* (pp. 255–269). Preprint on IACR ePrint 2008/152.

[7] Barthe, G., Danezis, G., Grégoire, B., Kunz, C., & Zanella-béguelin, S. (2013). Verified computational differential privacy with applications to smart metering. In *In 26th IEEE Computer Security Foundations Symposium, CSF 2013, Los Alamitos.*

[8] Barthe, G., Fournet, C., Grégoire, B., Strub, P.-Y., Swamy, N., & Zanella-Béguelin, S. (2014). Probabilistic relational verification for cryptographic implementations. In *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014*: ACM. To appear.

[9] Barthe, G., Grégoire, B., Heraud, S., & Zanella Béguelin, S. (2011a). Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science* (pp. 71–90).: Springer.

[10] Barthe, G., Grégoire, B., Lakhnech, Y., & Zanella Béguelin, S. (2011b). Beyond provable security verifiable ind-cca security of oaep. In A. Kiayias (Ed.), *Topics in Cryptology (CT-RSA) 2011*, volume 6558 of *Lecture Notes in Computer Science* (pp. 180–196). Springer Berlin Heidelberg.

[11] Barthe, G., Grégoire, B., & Zanella Béguelin, S. (2009). Formal certification of code-based cryptographic proofs. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009* (pp. 90–101).: ACM.

[12] Barthe, G., Köpf, B., Olmedo, F., & Zanella Béguelin, S. (2012). Probabilistic relational reasoning for differential privacy. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12 (pp. 97–110). New York, NY, USA: ACM.

[13] Bellare, M. (2006). New proofs for NMAC and HMAC: Security without collision-resistance. In *Advances in Cryptology (CRYPTO) 2006* (pp. 602–619). Springer.

[14] Bellare, M., Canetti, R., & Krawczyk, H. (1996). Keying hash functions for message authentication. In *Advances in Cryptology (CRYPTO) 1996* (pp. 1–15).: Springer.

[15] Bellare, M. & Rogaway, P. (2004). Code-based game-playing proofs and the security of triple encryption. Cryptology ePrint Archive, Report 2004/331. `http://eprint.iacr.org/`.

[16] Bengtson, J., Bhargavan, K., Fournet, C., Maffeis, S., & Gordon, A. D. (2008). Refinement types for secure implementations. In *In 21st IEEE Computer Security Foundations Symposium (CSF 08)* (pp. 17–32).: IEEE.

[17] Berg, M. (2013). *Formal Verification of Cryptographic Security Proofs*. PhD thesis, Saarland University.

[18] Bhargavan, K., Fournet, C., Kohlweiss, M., Pironti, A., & Strub, P. (2013a). Implementing tls with verified cryptographic security. In *Security and Privacy (SP), 2013 IEEE Symposium on* (pp. 445–459).

[19] Bhargavan, K., Fournet, C., Kohlweiss, M., Pironti, A., & Strub, P. (2013b). Implementing tls with verified cryptographic security. In *Security and Privacy (SP), 2013 IEEE Symposium on* (pp. 445–459).: IEEE.

[20] Blanchet, B. (2007). Computationally sound mechanized proofs of correspondence assertions. In *20th IEEE Computer Security Foundations Symposium (CSF'07)* (pp. 97–111). Venice, Italy: IEEE.

[21] Blanchet, B., Jaggard, A. D., Scedrov, A., & Tsay, J.-K. (2008). Computationally sound mechanized proofs for basic and public-key Kerberos. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS'08)* (pp. 87–99). Tokyo, Japan: ACM.

[22] Brown, D. R. L. & Gjøsteen, K. (2007). A security analysis of the nist sp 800-90 elliptic curve random number generator. Cryptology ePrint Archive, Report 2007/048. `http://eprint.iacr.org/`.

[23] Cadé, D. & Blanchet, B. (2013a). From computationally-proved protocol specifications to implementations and application to SSH. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, 4(1), 4–31. Special issue ARES'12.

[24] Cadé, D. & Blanchet, B. (2013b). Proved generation of implementations from computationally secure protocol specifications. In *Principles of Security and Trust* (pp. 63–82). Springer.

[25] Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.-C., & Steiner, M. (2013). Highly-scalable searchable symmetric encryption with support for boolean queries. In R. Canetti & J. Garay (Eds.), *Advances in Cryptology (CRYPTO) 2013*, volume 8042 of *Lecture Notes in Computer Science* (pp. 353–373). Springer Berlin Heidelberg.

[26] Damgård, I. B. (1990). A design principle for hash functions. In G. Brassard (Ed.), *Advances in Cryptology (CRYPTO) 1989 Proceedings*, volume 435 of *Lecture Notes in Computer Science* (pp. 416–427). Springer New York.

[27] Datta, A., Derek, A., Mitchell, J. C., & Roy, A. (2007). Protocol composition logic (pcl). *Electronic Notes in Theoretical Computer Science*, 172, 311–358.

[28] Datta, A., Derek, A., Mitchell, J. C., Shmatikov, V., & Turuani, M. (2005). Probabilistic polynomial-time semantics for a protocol security logic. In *Proceedings of the 32nd international conference on Automata, Languages and Programming*, ICALP'05 (pp. 16–29). Berlin, Heidelberg: Springer-Verlag.

[29] De Cristofaro, E., Jarecki, S., Liu, X., Lu, Y., & Tsudik, G. (2010). Privacy-protecting information retrieval, University of Irvine team: Protocol and proofs. Appendix E of SPAR Program BAA: `https://www.fbo.gov/utils/view?id=32750071e5cf4afc3b7e973d608e657e`.

[30] Elgamal, T. (1985). A public key cryptosystem and a signature scheme based on discrete logarithms. *Information Theory, IEEE Transactions on*, 31(4), 469–472.

[31] Fournet, C., Kohlweiss, M., & Strub, P.-Y. (2011). Modular code-based cryptographic verification. In Y. Chen, G. Danezis, & V. Shmatikov (Eds.), *ACM Conference on Computer and Communications Security* (pp. 341–350).: ACM.

[32] Goldreich, O. (2006). *Foundations of Cryptography: Volume 1*. New York, NY, USA: Cambridge University Press.

[33] Goldwasser, S. & Micali, S. (1984). Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2), 270 – 299.

[34] Halevi, S. (2005). A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181. `http://eprint.iacr.org/`.

[35] Herzog, J., Meadows, C., Jaggard, A., Stoughton, A., & Katz, J. (2013). MITLL-NRL panel: Easycrypt 0.2 feedback and opinions. `http://web.archive.org/web/20140703170052/https://easycrypt.info/trac/wiki/SchoolUPen2013`. Accessed: 2014-07-03.

[36] Leroy, X. (2009). Formal verification of a realistic compiler. *Communications of the ACM*, 52(7), 107–115.

[37] The Coq development team (2004). *The Coq proof assistant reference manual*. LogiCal Project. Version 8.0.

[38] Merkle, R. C. (1990). A certified digital signature. In G. Brassard (Ed.), *Advances in Cryptology (CRYPTO) 1989 Proceedings*, volume 435 of *Lecture Notes in Computer Science* (pp. 218–238). Springer New York.

[39] Nipkow, T., Paulson, L. C., & Wenzel, M. (2002). *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer.

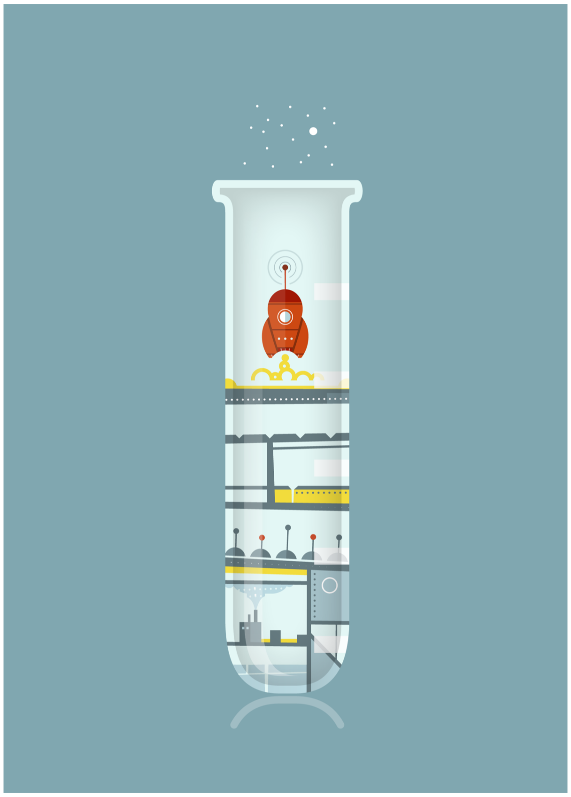
[40] Nowak, D. (2007). A framework for game-based security proofs. Cryptology ePrint Archive, Report 2007/199. `http://eprint.iacr.org/`.

[41] Ramsey, N. & Pfeffer, A. (2002). Stochastic lambda calculus and monads of probability distributions. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02 (pp. 154–165). New York, NY, USA: ACM.

[42] Roy, A., Datta, A., & Mitchell, J. (2008). Formal proofs of cryptographic security of diffie-hellman-based protocols. In G. Barthe & C. Fournet (Eds.), *Trustworthy Global Computing*, volume 4912 of *Lecture Notes in Computer Science* (pp. 312–329). Springer Berlin Heidelberg.

[43] Shumow, D. & Ferguson, N. (2007). On the possibility of a back door in the nist sp800-90 dual ec prng. Advances in Cryptology (CRYPTO) 2007, Rump Session Talk. `http://rump2007.cr.yp.to/15-shumow.pdf`.

[44] Stoughton, A. (2013). Mini-app: Proving a simple private information retrieval protocol secure in easycrypt. `http://web.archive.org/web/20140703170052/https://easycrypt.info/trac/wiki/SchoolUPen2013`. Accessed: 2014-07-03.



This thesis was typeset using LaTeX, originally developed by Leslie Lamport and based on Donald Knuth's TeX. The body text is set in 11 point Egenolff-Berner Garamond, a revival of Claude Garamont's humanist typeface. The above illustration, "Science Experiment 02", was created by Ben Schlitter and released under CC BY-NC-ND 3.0. A template that can be used to format a PhD thesis with this look and feel has been released under the permissive MIT (xii) license, and can be found online at github.com/suchow/Dissertate or from its author, Jordan Suchow, at suchow@post.harvard.edu.